A Speech Splicing System For Vocal Shaping

by

Kenneth Ferens

A Thesis

presented to the University of Manitoba

in partial fulfillment of

the requirements of the degree of

Master of Science

in

the Department of Electrical and Computer Engineering

Winnipeg, Manitoba

May, 1991

ISBN 0-315-76816-9

A SPEECH SPLICING SYSTEM FOR VOCAL SHAPING

BY

KENNETH FERENS

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

ABSTRACT

A speech splicing system is developed for automated vocal shaping. A joint study of vocal shaping and speech synthesis leads to a determination of optimal synthesis techniques, tools, and units, as well as a development of a speech processing system based on a PC AT for automated vocal shaping. Subjective tests are conducted in order to determine the effectiveness of the synthesis techniques, tools, and units. The adaptive differential pulse code modulation (ADPCM) technique of data compression is used in order to reduce the transmission rate of speech by one-half, while maintaining good toll quality. Coarse speech splicing is done with the copy, cut, and paste synthesis tools, while amplitude interpolation and linear predictive extrapolation (LPE) are used for fine adjustment of boundary properties. Isolated phoneme and extracted sub-word synthesis units are chosen in order to facilitate investigation of both small and large vocabulary needs. An external board (consisting of an Oki MSM6258VJS ADPCM speech processor, a dual-pointer FIFO buffer, and a 6802 μP) and a host computer (PC AT) are connected via an RS-232C compatible serial communications channel. The 6802 μP utilizes the FIFO buffer for controlling the asynchronous communication of speech data between the speech processor and host computer. Performed on the host computer, the speech processing software includes real-time disk recording and playing of speech, serial port initialization, time domain plot of PCM data, and selection of any portion of speech data for playing, copying, cutting, pasting, extrapolating, and averaging. Since the hard disk is used as virtual RAM, any size file can be processed (real-time recording duration limited by available hard disk space). Speech splicing experiments include library expansion by extraction, sub-word concatenation, and isolated phoneme concatenation. Preliminary results show that word synthesis by sub-word concatenation achieves up to 80% natural quality.

ACKNOWLEDGEMENTS

It is a pleasure to acknowledge the people who helped me throughout the course of this thesis. First and foremost, I would like to express my appreciation and admiration for my advisor, Dr. W. Kinsner. Your gift of dispensing motivation and inspiration in times of need turns trees into forests, night into day, and indifference into fiery purpose. Thank you also for the thesis topic.

I would also like to thank Chris Love, Janice Miller, Armein Langi, and Adi Indrayanto for their careful reading of the manuscript, support, and friendship. For their technical expertise, I would like to thank the Electrical Engineering technicians, in particular, Ken Biegun, who helped me with the design of the FIFO buffer.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ACIA	Asynchronous Communications Interface Adapter
ADC	Analog to Digital Converter
ADPCM	Adaptive Differential Pulse Code Modulation
AI	Amplitude Interpolation
B	Byte
BIOS	Basic Input Output System
bps	Bits Per Second
Byte	Eight Binary Bits
CASS	Computer Automated Speech Splicing
CPU	Central Processing Unit
CGA	IBM's Color Graphics Adapter
DAC	Digital to Analog Converter
DOS	Disk Operating System
DRAM	Dynamic Access Memory
EGA	Enhanced Graphics Adapter
EPROM	Erasable Programmable Read Only Memory
FIFO	First-In-First-Out
GUI	Graphical User Interface
HGC	Hercules Graphics Adapter
IC	Integrated Circuit
IEEE	The Institute of Electrical and Electronics Engineers, Inc
I/O	Input/Output
IPA	International Phonetic Alphabet
Hz	Cycles per second
k	1000
K	$2^{10} = 1024$
LPC	Linear Predictive Coding
LPE	Linear Predictive Extrapolation
MCGA	Multicolor Graphics Array
M	$2^{20} = 2^{10}2^{10} = 2^{10}K = 1048576$ (Binary context)
M	1 000 000 (Decimal context)
MPU	MicroProcessor Unit
µsec	MicroSecond
µP	MicroProcessor
Nibble	Four Binary Bits
nsec	NanoSecond
PCM	Pulse Code Modulation
PIA	Peripheral Interface Adapter
ROM	Read Only Memory
RS-232	Recommended Standard #232
SIMM	Single Inline Memory Module
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver Transmitter
VDU	Video Display Unit
VGA	Video Graphics Adapter

CHAPTER I
INTRODUCTION

1.1 Purpose

The purpose of this thesis is to present a computer aided speech splicing (CASS) system for vocal shaping. CASS is a tool intended to assist speech therapists in the vocal shaping process.

1.2 Problem

Speech is the most widely used form of communication. We take for granted our ability to express ourselves through language of the spoken word. To some individuals, in particular, retarded and autistic persons, intelligible speech does not come easily and may not even be possible. This must be very frustrating, discouraging, and disheartening.

Much research has been done on methods of teaching speech to voice-handicapped individuals. One method involves the use of psychological shaping. "Shaping is a behavioral procedure that has been used to develop or train a wide variety of new behaviors in both animals and humans" [Cair90]. Conventionally, vocal shaping involves a direct interaction between speech therapist and student. The speech therapist provides the example, typical model, or prototype target response, and the student attempts to reproduce it. After each trial, the therapist assesses the quality and determines the "errors in articulation such as omissions, substitutions, additions, or distortions of speech sounds" [Desr90]. Based on certain criteria, the response is judged as either progressive or regressive. Following an improved response, reinforcement is administered, e.g., by saying

'very good' or by giving the student a food reward.

In order to avoid the subjectivity involved in assessing errors in articulation, much research has been done in automated vocal shaping systems [Cair90], [PeKR87], [Desr90], [FlHa83], [KWMR87], and [Perk71]. In these systems, errors in articulation were measured in terms of a distance (e.g., Euclidean) between the student's response and the target sound. The system decided whether the response was close enough "...within a criterion region..." [PeKR87] to warrant dispensing of reinforcement or to prompt another trial.

Most of this past research was limited to shaping fundamental units of speech, called phonemes, such as /A/, /AE/, and the phone 'ah'. Some success was reported: "Human ratings indicated that all three children showed some improvement in their ability to imitate the trained phonemes" [Cair90]. Also, it was reported that "The correlation of the judgements made by the apparatus (automated vocal shaping system) with each speech professional was high, although not as high as the correlation between the professionals themselves" [PeKR87].

Given the success of shaping phonemes, a natural extension to this work is to shape new words or phrases formed by integrating two or more shaped phonemes. There are two methods in which this may be accomplished. One method, which is similar to phoneme target acquisition, requires that the therapist supplies the pronunciation of the new words or phrases. A possible problem associated with this method is the introduction of an uncontrolled experimental variable. Vocal shaping requires continuity, perseverance and years for its full and complete implementation. The same therapist may not be available for prolonged or even short periods of time. Furthermore, throughout the process of shaping, the student may become accustomed to the voice of an individual therapist. The introduction of a different voice, a different pronunciation of the same target sound, may set

back past accomplishments and delay future advancement while the student becomes comfortable with the new teacher. It is difficult to control a scientific experiment while allowing the introduction of new variables whose effects are not fully understood.

Another method of vocabulary expansion involves using, as a set of target words, those words formed by an electronic concatenation of a set of previously stored and smaller units of speech spoken by a therapist. The stored set of smaller units of speech would form a basis upon which longer words or phrases may be formed. In this way, the new set of words would be inherently characteristic of the original therapist's voice. The new words may resemble the pronunciation of the original therapist even though their production would not be entirely vocalized.

Another possible benefit of employing electronic concatenation for vocal shaping is realized by taking the idea of voice familiarization one step further. While using a therapist's voice as the target word may have some success, a different approach involves using a target word formed by a concatenation of modified versions of the student's learned phonemes themselves. Rather than trying to emulate a teacher's voice, the student would be training his/her voice as it would sound were the pronunciation or vocalization correct.

In order to accomplish electronic concatenation for vocal shaping, a system is needed which is capable of splicing isolated phonemes or extracted sub-word units to form naturally sounding new words, utterances, or phrases. This is one of the problems considered in this thesis. However, there are other induced issues.

An analog method of splicing electronic information, such as editing audio and visual tape, would not be possible nor practical for speech splicing. On the one hand, serious speech splicing entails modification of the speech waveform itself, and this is not

possible by simply cutting and pasting tape. However, if vocal shaping is limited to training entire words or combinations thereof, then simple cutting and pasting may be sufficient. On the other hand, locating specific speech information on tape, cutting a portion of it, and then pasting this information at a carefully selected location is an arduous and tedious task, and, therefore, it is not practical.

A digital method of speech processing is a valid alternative. However, sound quality issues arise. For example, the presentation of poor reproductions of speech would probably be unrepresentative and lead to indifference. On the other hand, very good quality speech has a cost associated with transmission rate, bandwidth, and computer memory. Past work done by [KlKi87] (see Fig. 2.3) clearly shows the trade-off between speech quality and the above mentioned costs. The middle ground must be chosen such that speech quality is acceptable, while the cost of transmission rate, bandwidth, and computer memory is low.

1.3 Scope

This thesis consist of seven chapters. Chapter I states the purpose of the thesis, discusses the major problems to be solved by the thesis, and provides some motivation for the thesis. Chapter II gives background information on the thesis. In this chapter a review of speech synthesis is followed by a review of vocal shaping, and it is shown how certain speech synthesis concepts may be applied to vocal shaping systems. As such, this chapter lays out the theoretical and psychological aspects of this thesis. This provides motivation for the technical aspect, which is discussed in Chapter III. Chapter III provides a block diagram description of the speech splicing system. In this chapter, it is shown where and how speech splicing fits into the vocal shaping environment. This chapter is also intended as an introduction to Chapter IV, which is a detailed description of the speech splicing system. This chapter and Chapter V may be skipped without any significant loss of continuity. However, while these chapters are geared for digital hardware and software

designers, i.e., Electrical and Computer Engineers, the main ideas are still understandable by people from other disciplines. Chapter V offers an alternative paper design of a buffer for a speech splicing system. This design uses a different technology than that used by the buffer described in Chapter IV. Chapter V is intended for comparison purposes, and it illustrates the fact that there is more than one way to design a buffer. Chapter VI discusses the speech splicing experiments. These experiments utilize the speech synthesis methods discussed in Chapter II and the speech splicing system of Chapter IV in order to show how speech splicing can be used as a tool for automated vocal shaping systems. Finally, Chapter VII gives the conclusions and recommendations.

CHAPTER II
BACKGROUND ON SPEECH SYNTHESIS/VOCAL SHAPING

This thesis deals with applying speech synthesis techniques to automated vocal shaping systems. In particular, certain compression, analysis, and waveform concatenation techniques are used in order to aid vocal shaping systems. As such, this chapter briefly reviews the pertinent areas of speech synthesis and vocal shaping. Furthermore, this chapter introduces the theory and methods of three waveform concatenation techniques capable of synthesizing new words by concatenating isolated phonemes or sequences of phonemes extracted from previously recorded words. These three waveform concatenation techniques are used in Chapter VI Speech Splicing Experiments.

2.1 Review of Speech Synthesis

This section provides a brief review of speech synthesis. While the area of speech synthesis is vast, this review is not meant to cover all the areas, but it is intended to define the scope and to focus on the compression, analysis, and waveform concatenation techniques used in this thesis. In particular, the compression technique of adaptive differential pulse code modulation (ADPCM), the analysis and synthesis technique of linear predictive coding (LPC), and three waveform concatenation techniques are described in more detail.

2.1.1 Building Blocks of Speech

A study of speech synthesis usually begins with analyzing the construction of human speech. One way of understanding how a system is constructed is to break it down

and study its component parts, ideally the smallest building blocks. By analysing the building blocks of speech and uncovering how these blocks are put together and integrated in order to form the whole, we may be able to synthesize speech. In order to study these building blocks, a language for unambiguously and exactly describing them is required.

2.1.1.1 Phones, Phonemes, and Allophones

The established science of describing the building blocks of speech through language has a well developed theoretical background. The International Phonetic Alphabet (IPA) is a system that describes speech sounds. The two branches of this science are (i) phonetics and (ii) phonemics. Phonetics deals with providing a one-to-one mapping from every known human speech sound to a written text representation. On the other hand, phonemics deals with providing a written text representation of the fundamental or smallest units of speech of a particular language. In phonetics an individual sound is called a phone, whereas, in phonemics the smallest unit is called a phoneme. Phones are transcribed by placing the text description between square brackets (e.g., [æ]), while phonemes are placed within slants (e.g., /IY/).

Phonemes

Phonemes are actually a subset of phones. However, a phone is a phoneme in a particular language if it changes the meaning of a word upon replacing another phoneme in that word. For example, in English the words 'top' and 'cop' differ in the first phoneme, i.e., /T/ and /K/, respectively. The phonemes /T/ and /K/ are said to be phonemic in English. Table 2.1 shows some of the 47 phonemes in the English language [Pars86].

Allophones

As a counter example, consider the vowels in words 'coat' and 'coal'. While the substitution of one vowel for the other may result in a strange pronunciation, the vowels are

not considered as two different phonemes because the meaning of the words do not change if the vowel sounds are interchanged. These two vowels are referred to as allophones of the phoneme /OW/.

Table 2.1 Phonetic transcription of some General American English Phonemes (adapted from [Pars86, p. 85]).

Vowels

IPA	Typewritten	Example	IPA	Typewritten	Example
i	IY	h*ee*d	o	OW	h*oe*d
I	IH	h*i*d	U	UH	h*oo*d
e	EY	h*ay*ed	u	UW	wh*o*'d
ɛ	EH	h*ea*d	Λ	AH	b*u*d
æ	AE	h*a*d	a	AA	h*o*d

Consonants

IPA	Typewritten	Example	IPA	Typewritten	Example
p	P	*p*op	t	T	*t*ell
tʃ	CH	*ch*eek	k	K	*c*ool
b	B	*b*at	d	D	*d*oor
dʒ	JH	*j*ust	g	G	*g*irl
f	F	*f*ight	θ	TH	*th*ick
s	S	*s*ick	ʃ	SH	*sh*ock
h	HH	*h*at	v	V	*v*eal
ð	DH	*th*at	z	Z	*z*eal
ʒ	ZH	mea*s*ure	m	M	*m*at
n	N	*n*ose	ŋ	NX	ba*ng*
l	L	c*a*ll	r	R	*r*ide
j	Y	*y*et	w	W	*w*et

2.1.1.2 Distinguishing Features

In order to better understand how speech may be constructed, it is beneficial to identify features of phonemes that may be used to distinguish them from others. Phonemes may be distinguished on the basis of psychological perception, on the basis of objective properties of speech, or on a combination thereof.

We are all familiar with vowel and consonant classifications of phonemes. An objective feature of vowels is that their production is characterized by an unconstricted flow of air through the vocal tract. This results in vowels generally exhibiting a pseudo-periodic sound. On the other hand, consonants are characterized by a constriction in the vocal tract, and this results in a noise-like sound. While most vowels and consonants may be distinguished by the constriction criterion, there are examples where the distinction is fuzzy. That is, there are some consonants that exhibit a degree of periodic sounds and a degree of constriction in the vocal tract, for example, /l/ in 'call'. For systems relying on distinguishing vowels and consonants based on the constriction criterion, it may be beneficial to employ the theory of fuzzy logic [FeKi91].

From an objective viewpoint, phonemes may be differentiated with respect to their frequency content. Fig. 2.1 shows a plot of frequency F1 vs. F2 for phonemes spoken by various people. The frequencies F1 and F2 are called formant frequencies, and they

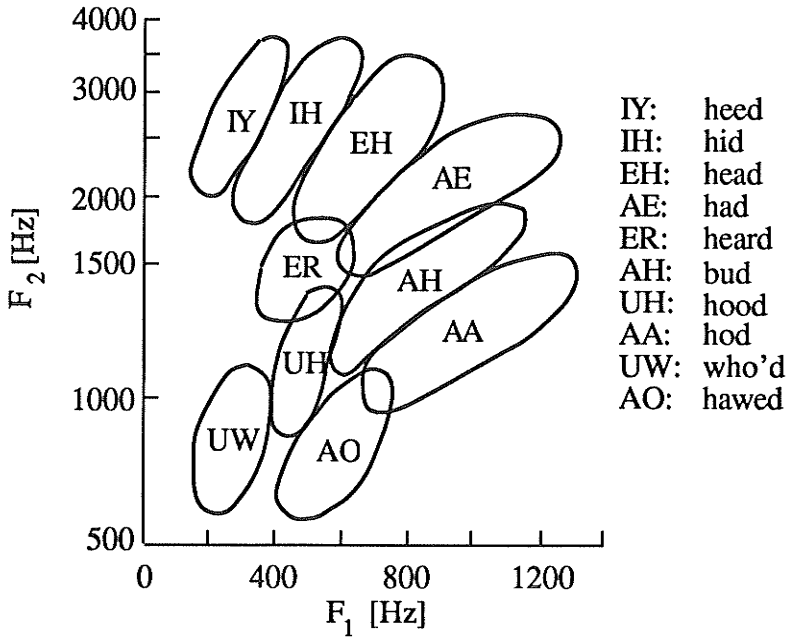


Fig. 2.1 Frequency distinction of vowels (after [Pars86, p. 105]).

represent those frequencies having the largest power of the particular phoneme. The orthogonal information in this figure may be used in a limited phoneme recognition system. In particular, phonemes /IY/ and /UW/ may be distinguished directly by examining the relative contribution of frequency bands from 500 to 1200 Hz and from 2000 to 3700 Hz. However, as shown in the figure, there are overlapping regions where this method may not work.

2.1.1.2.1 Coarticulation

One misconception and unfortunate fact of speech production is that when we form a word, we do not simply generate phonemes in isolation and put them together. If it were that simple, speech synthesis would not be a problem as it is today. When we utter a word, each phoneme in that word is influenced to some extent by its neighbors. We can think of each phone in a word "as a *target* at which the vocal organs aim but which they never reach. As soon as the target has been approached nearly enough to be intelligible to the listener, the organs change their destinations and start to head for a new target. This is done to minimize the effort expended in speaking and makes for greater fluency" [Pars86, p. 92]. Not only are phonemes influenced by their predecessors, but the predecessors, themselves, are influenced by the following phonemes. Figure 2.2 shows an example of the coarticulation phenomenon. The top figure shows a plot of frequency vs. time (called a sonogram) of naturally spoken word 'feet', while the bottom figure shows the sonogram of synthetically formed 'feet'. Note the continuous, smooth, and gliding change in frequency between the adjacent 'f' and 'e' sounds in naturally spoken 'feet', and contrast this to the discrete and abrupt change between that for synthetically formed 'feet'. The influence and overlapping of properties of individual phonemes is called coarticulation.

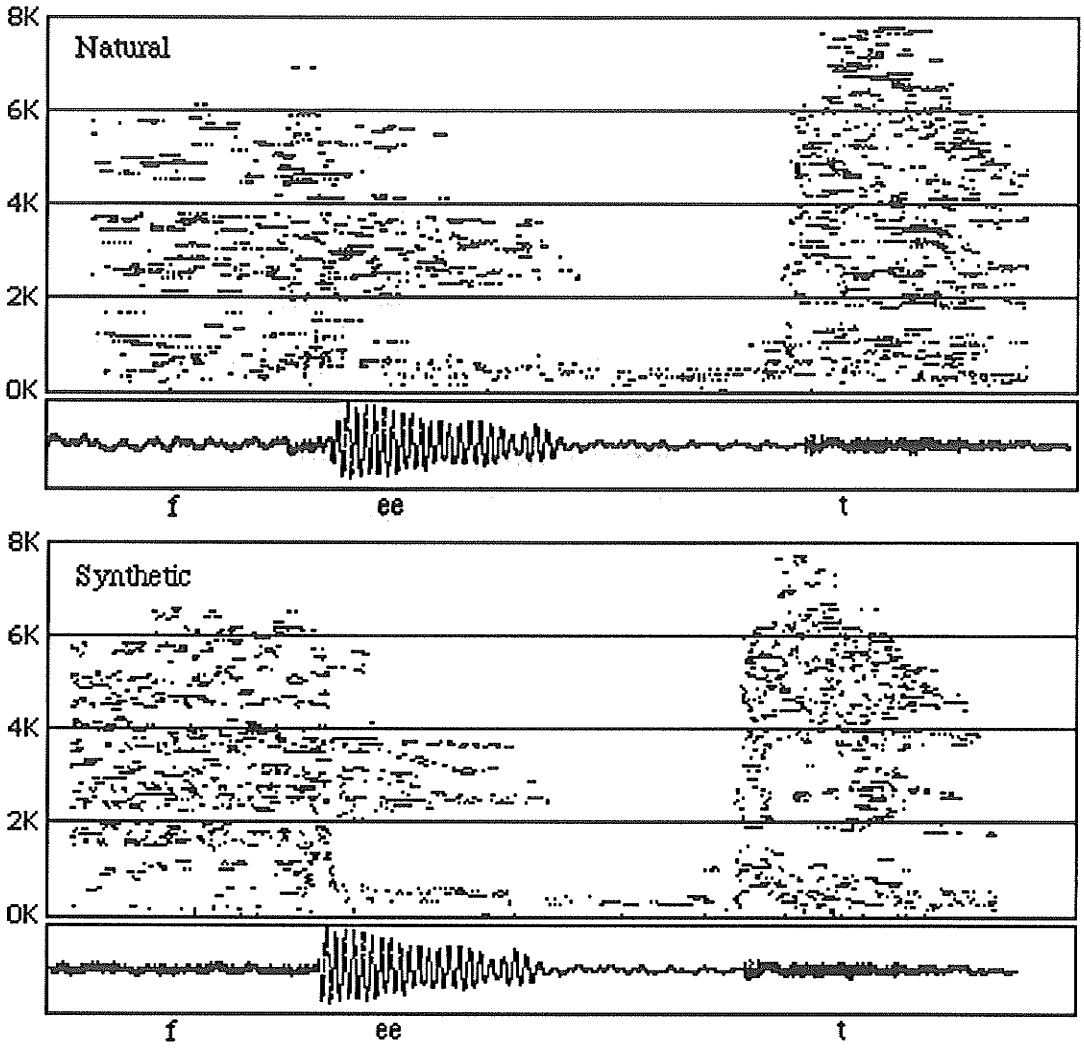


Fig. 2.2 Sonograms of 'feet'. The top plot shows the sonogram for naturally spoken 'feet', while the bottom shows the sonogram for synthetically produced 'feet'.

It is evident that if a speech synthesis system based on phoneme construction is to be successful and synthesize naturally sounding speech, rules or methods are required in order to take coarticulation into account. There are different methods which can be used to deal with coarticulation, and they all work with varying degrees. These methods are incorporated in the general speech synthesis methods, namely, synthesis by rule, synthesize by analysis, digital recording, and waveform synthesis. This thesis specifically deals with waveform synthesis, but the others are briefly reviewed in order to develop scope.

2.2 Speech Synthesis Methods

Speech synthesis deals with reproducing human voice from an electrical and/or mechanical representation. One of the earliest speech synthesizers was developed in 1947 at Haskins Laboratory [Bris84], where speech was first captured and represented in the form of a spectrograph and then later played back. Today, there are many different speech synthesizers, but they all may be classified, as shown in the middle of Fig. 2.3, as either synthesis by rule, synthesis by analysis, or digital recording.

As shown in Fig. 2.3, each synthesis method is associated with a cost trade-off between quality and transmission rate: the higher the quality, the higher is the transmission rate. The designer of a speech synthesizer must consider what quality of speech is required and whether the bandwidth of the system is capable of accommodating the associated transmission rate. For example, in many telephone systems, the existing communications channel, i.e., the wires, cables, and repeaters connecting one telephone customer to the next, has a capacity of allowing transmission of electrical signals of frequency up to about 4 000 Hz (32 000 bits/sec if each sample of data is represented by eight bits). This is why toll quality is the best possible in telephone systems. In this thesis toll quality is chosen in the vocal shaping system, since toll quality is sufficiently intelligible for the general public and because the bandwidth of computer systems generally allow the associated transmission rate.

2.2.1 Synthesis by Rule

One of the primary objectives and well known purposes of speech synthesis is to convert printed text into understandable sound messages. Much like written text conveys meaning by stringing together discrete symbolics units, text to speech synthesizers

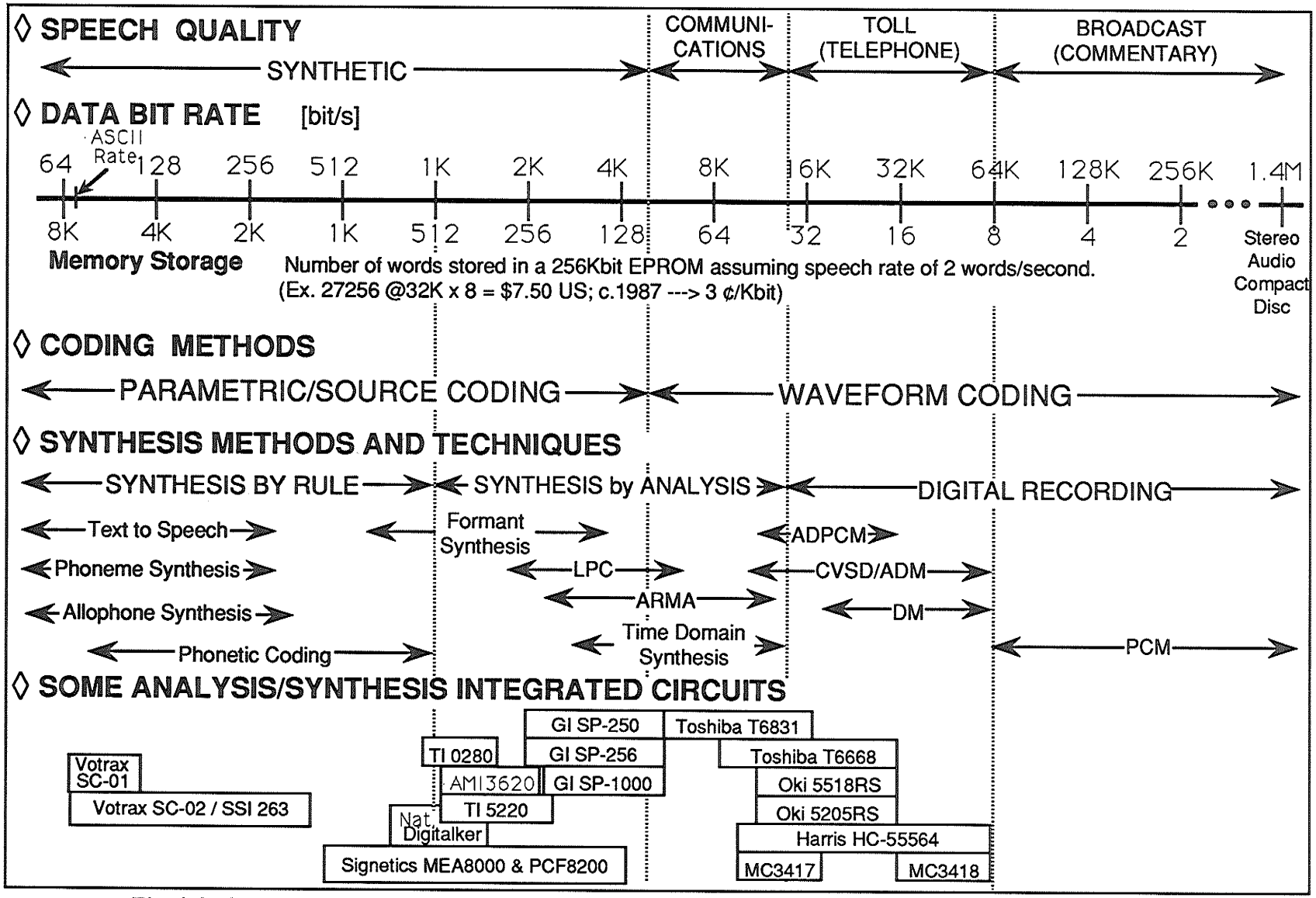


Fig. 2.3 Comparison of speech coding techniques, bit rates and associated speech quality. (after [KlKi87]).

concatenate sequences of sounds to form words and sequences of words to form phrases and sentences. Furthermore, the assembly of words, phrases, and sentences is controlled by a set of rules, which, not unlike written text, is language specific.

Much research goes into studying the rules that control how sounds are put together to form words, phrases, and sentences [Sagi90]. Synthesizing speech from arbitrary text input involves highly advanced information processing, including text analysis, syntactic analysis, pronouncing dictionary, accent assignment, prosody control, segment duration, rhythm, tone, and loudness.

2.2.2 Synthesis by Analysis

Speech can be reproduced from an analytical representation. The frequency domain and time domain of actual speech may be analyzed in order to determine characteristic features and redundancy. These characteristics are used to form a compressed representation. The function of the synthesizer is to decode the compressed representation and put back together the speech. During the course of compression, some information will be lost and, consequently, the quality will suffer. The compression ratio and the resulting quality are criteria used to judge the goodness of a particular technique.

The frequency domain analysis/synthesis technique is based on a model of the human vocal system. This model generally consists of excitation sources (modelling the vocal cord oscillations, turbulent air, and the lung) and a variety of filters (modelling the acoustic filter of the vocal tract). Data compression is achieved by storing and transmitting parameters of the model instead of the original waveform. The number and frequency of parameters is smaller than the amount of information in the waveform because speech is pseudo periodic and redundant. The synthesizer uses the parameters in order to reconstruct

the speech waveform. The output waveform may not bear any detailed resemblance to the original waveform, and yet the quality may still be quite intelligible and resemble the sound of the original speech.

While frequency-domain techniques achieve compression through representations of characteristic features of the speech waveform, time-domain techniques, in contrast, achieve compression through compressed representations of the time-domain waveform itself. However, similar to frequency-domain techniques, time-domain techniques achieve compression by exploiting the pseudo-periodic and redundant nature of speech. The synthesizer's job is to de-compress the representation and reproduce the time domain waveform. Similar to the frequency-domain synthesizer, the reproduced time-domain waveform need not be an identical match of the original waveform.

2.2.2.1 Linear Predictive Coding (LPC)

One of the more important analysis/synthesis methods is called linear predictive coding (LPC). LPC provides a mathematical model of a linear, discrete-time system. Human speech production may be modelled as an all-pole linear, discrete-time system. Eq. 2.1 shows the mathematical expression used by LPC to model the speech waveform.

$$\hat{y}[n] = -\sum_{i=1}^{p} a[i]y[n-i] \tag{2.1}$$

As this equation indicates, LPC predicts the forthcoming time domain sample of the speech waveform by calculating a linear combination of past samples [Pars86]. The hat over the y indicates an estimate and p represents the number of past samples. The quantities, a[i], are called predictor coefficients and are determined by minimizing the mean-squared error given by Eq. 2.2. Eq. 2.2 yields p equations, which can be solved for a[i]. The

$$\min(\text{Error}) = \min\left\langle (y[n]-\hat{y}[n])^2 \right\rangle \tag{2.2}$$

computation of a[i] usually involves either the autocorrelation or the covariance method. It has been found that the autocorrelation method is more suitable for stationary segments of speech, while the covariance is better suited for non-stationary segments [Pars86]. Note that Eq. 2.1 can also be viewed as a digital filter, where a[i] is the impulse response of the filter. In this respect, Eq. 2.1 is a convolution operation.

LPC is also useful for estimating a preceding sample of stationary speech. For example, given p samples, y[1] through y[p], the preceding sample, y[0], can be estimated or postdicted by Eq. 2.3. Note that the only difference between Eq. 2.1 and Eq. 2.3 is the direction of the data in the convolution. This is a result of the assumption of stationary speech, which implies the statistics do not take the direction of time into consideration.

$$\hat{y}[n] = -\sum_{i=1}^{p} a[p+1-i]y[n+i] \tag{2.3}$$

Finally, LPC achieves compression by storing and transmitting the predictor coefficients, whose number is a fraction of the *p* data samples.

2.2.3 Digital Recording

Among the three synthesis techniques mentioned, digital recording offers the best quality. Like synthesis by analysis, synthesis by digital recording strives to achieve data compression. However, most of the compression techniques are implemented at the lower end of the digital recording spectrum, as shown in Fig. 2.3. At the high end, the quality is excellent, but the cost of the associated transmission rate can be quite high. For example, about 10 MBytes ([1.4 Mbits/sec][60 sec]/[8 bits/Byte]) of computer memory is required

in order to store one minute of compact disk quality sound. Like the analytical synthesizer, the function of the digital synthesizer is to decode the digital representation and reconstruct the speech signal. One difference is that the reproduced waveform resembles the original. In fact, the pulse code modulation (PCM) technique (no compression) is guaranteed to faithfully reproduce the waveform exactly (if we neglect quantization error) [FeLo89].

The digital recording method of speech synthesis actually requires two phases, (i) analysis and (ii) synthesis, for its complete implementation. The analog speech waveform is first represented in the digital domain by a process called digitization. As shown in Fig. 2.4, the waveform is quantized in both time and amplitude. That is to say, a continuous, electrical analog representation of speech is sampled at regularly spaced intervals (time quantization), and then the samples are rounded off to the nearest digital number (amplitude quantization). The resulting digital representation is stored in computer memory for further processing and eventual transmission for the synthesis phase. The digital synthesizer reconstructs the analog waveform from the digital representation. Synthesis is the reverse process of digital analysis. This exact reproduction capability implies that digital recording preserves all information in speech, such as pitch, intonation, inflection, and stress.

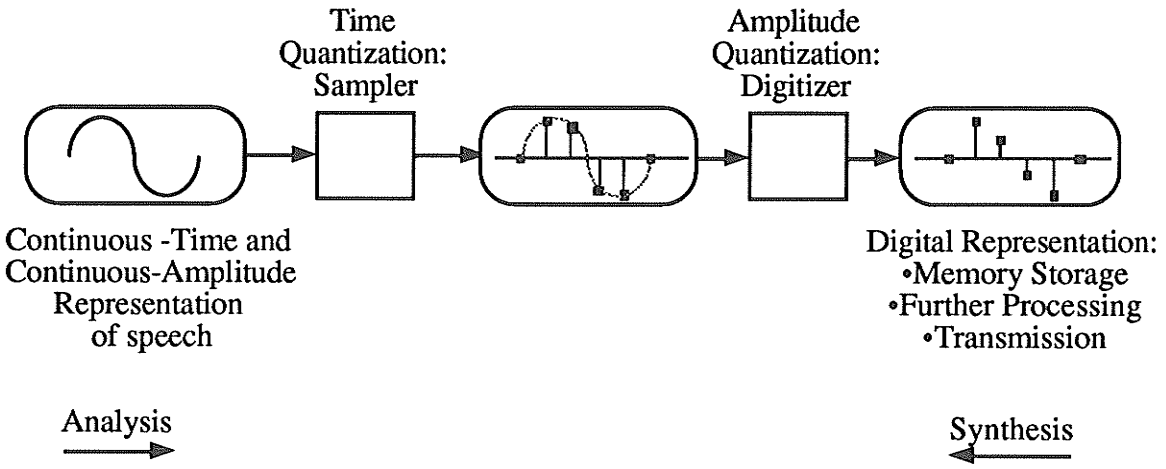


Fig. 2.4 Digital speech analysis and synthesis.

2.2.3.1 Adaptive Differential Pulse Code Modulation (ADPCM)

Adaptive differential pulse code modulation (ADPCM) is a data compression technique that is implemented at the lower end of the digital recording spectrum of speech (i.e., from about 12 kbps to 32 kbps). ADPCM is an adaptive and differential derivative of PCM. Rather than storing and transmitting the absolute value of each digitized sample, as is done in PCM, the difference between successive samples is taken, and then this difference signal is quantized. The differential quantization step size is adaptively derived from the relative size of the previous sample. The logic of operating on the difference signal rather than the absolute value of each sample is motivated by the fact that speech is inherently redundant [FeLo89]. That is to say, there is a high probability that successive amplitudes of speech samples are approximately the same. As a result, the size of data needed to encode the difference signal can be reduced. The strategy of the adaptive quantization step size is based on the fact that the amplitude of speech can be described by a Laplacian distribution [JaNo84]. Small amplitudes of speech have a relatively high probability, while the probability decreases significantly with higher amplitudes. Based on these statistical facts, the strategy is to use a large step size when the amplitude of the previous sample is large and small step size when the previous amplitude is relatively small. As a result, better resolution is provided when the amplitude is more probable. This strategy enables ADPCM to better track or match the variance of the input speech waveform.

2.2.4 Waveform Synthesis

Perhaps, the most straightforward way of synthesizing speech is to have digital recordings of individual speech units and to design a system for retrieving and stringing together those units at the correct time and the proper order. These units may be the actual digitally recorded waveform of large speech utterances, such as whole words or phrases or smaller utterances, such as phonemes, allophones, or phones. The type of synthesis unit

used by the synthesizer depends on the application. Small vocabulary systems may find it more suitable to use word waveforms, while waveforms of phonemes would be required for larger vocabulary systems. For example, a city bus schedule information system delivered over telephone lines would require only a limited vocabulary. A typical response is "Route 60, Pembina. Next bus at 2:10". The only words that may vary in this message are the number of the route (60), the name of the route (Pembina), and the time (2:10).

Larger vocabulary systems are increasingly more complicated. At the very extreme, we can imagine an unlimited, speaker independent system. In this huge system, the sound units would ideally consist of fundamental building blocks of speech, i.e., phonemes, allophones, and/or phones.

Large vocabulary systems must consider the coarticulation problem (refer to Chapter II, Section 2.1.1.2.1). Human speech (e.g., the formation of an individual word) does not merely consist of concatenating isolated phonemes. Each phoneme is influenced by its neighbors. Thus, if we were to string together phonemes uttered in isolation in order to form a new word, that word would most likely sound unnatural. A phoneme based system would have to either modify the boundary conditions of adjacent phonemes or, perhaps, increase the library size by including allophones or specific phones.

Indeed, the latter has been attempted by Harris [Harri53]. However, Harris realized that an immediate problem is the exponentially increasing size of memory required to store waveforms of different allophones for each phoneme. To solve the memory size problem, Harris investigated a minimal set of allophones.

Another way of decreasing the size of memory required by allophone based systems is to store larger units of speech that include the transition regions. These larger units of

speech are called dyads, diphones, or demisyllabes [PeWa58] and [Holm88], and they consist of phonemes and their transition regions. These transition regions are generally characterized as steady state regions, and they are not greatly influenced by adjacent sounds. Synthesis by diphones has been attempted by Peterson and Wang [PeWa58], but a problem they encountered was the discontinuity between diphones themselves.

It appears that if phoneme based synthesis for large or even small vocabularies is to succeed, then it must deal with the coarticulation problem, directly. The boundary properties of phonemes spoken in isolation must be modified before they can be put together to form larger utterances. This thesis investigates three methods of dealing with the boundary modification of isolated phonemes and extracted sub-word synthesis units. The process of forming new utterances by putting together smaller units and modifying their boundary properties is defined in this thesis as speech splicing. The following sections give the theory, requirements, and motivation of the proposed boundary property modifications.

2.2.4.1 Cut, Copy, and Paste

Perhaps, the simplest way of constructing speech from individual components is cutting, copying, and pasting digitally recorded waveforms. This method requires an interactive editing system capable of allowing the user to display, select, and playback any portion of a previously digitally recorded waveform. The displaying part of the system requires a visual association of a waveform with its sound. This capability provides great flexibility in selecting the right segment of speech for the job.

This method may also be suitable for modifying the items of small vocabulary systems, such as a bus schedule information service, as mentioned above. Figure 2.5 shows an example of a three step procedure for replacing the word 'ten' from the message "Route 60, Pembina. Next bus at 2:10." with the word 'twelve'. Both words have been previously

digitally recorded, and all that is seemingly required is to cut, copy, and paste the words at appropriate times or locations in the message.

However, this method is somewhat tedious, labour intensive, and time demanding, because the user must continually select and play portions of the time domain representation of the speech waveform, until the sound of interest has been located, isolated, and copied. This problem arises because the boundaries of phonemes located in an utterance are not clearly defined. Words spoken by humans involve a continuous flow or glide between adjacent phonemes. As such, specifications of the boundaries may well be fuzzy concepts and better treated by fuzzy theory [FeKi91].

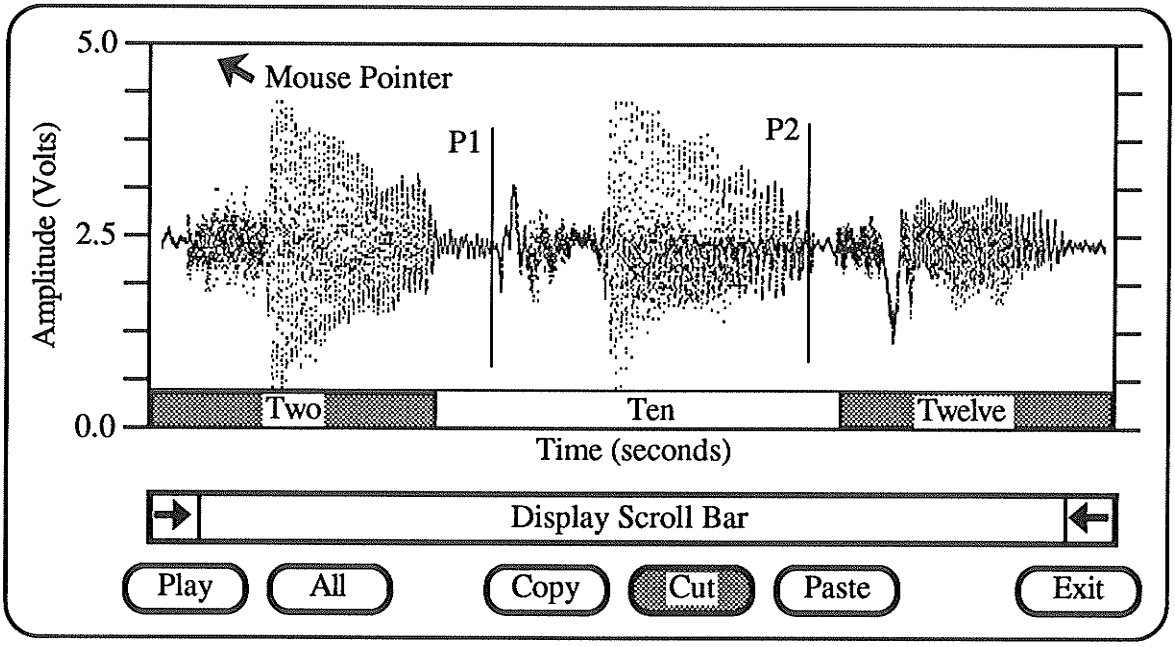


Fig. 2.5a Procedure for cutting waveform of word 'ten'. The waveform of the word 'ten' may be isolated by selecting a portion of it through pointers P1 and P2. Also, the visual representation of the waveform may be associated with the sound of the word 'ten' by playing back the selection. Finally, the selected waveform is cut by selecting cut function through the icon, *Cut*.

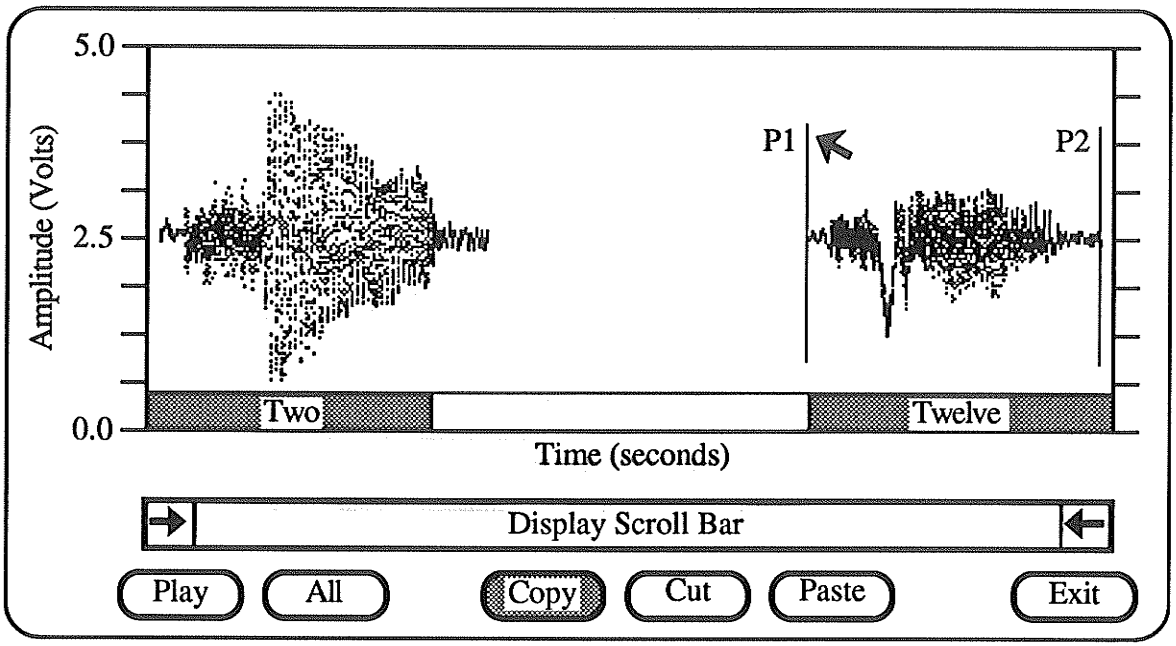


Fig 2.5b Procedure for copying the waveform of the word 'twelve'. Verification of the waveform by playing back the speech selected through pointers P1 and P2 is followed by using the icon, Copy, to temporarily retain a copy of the selected waveform.

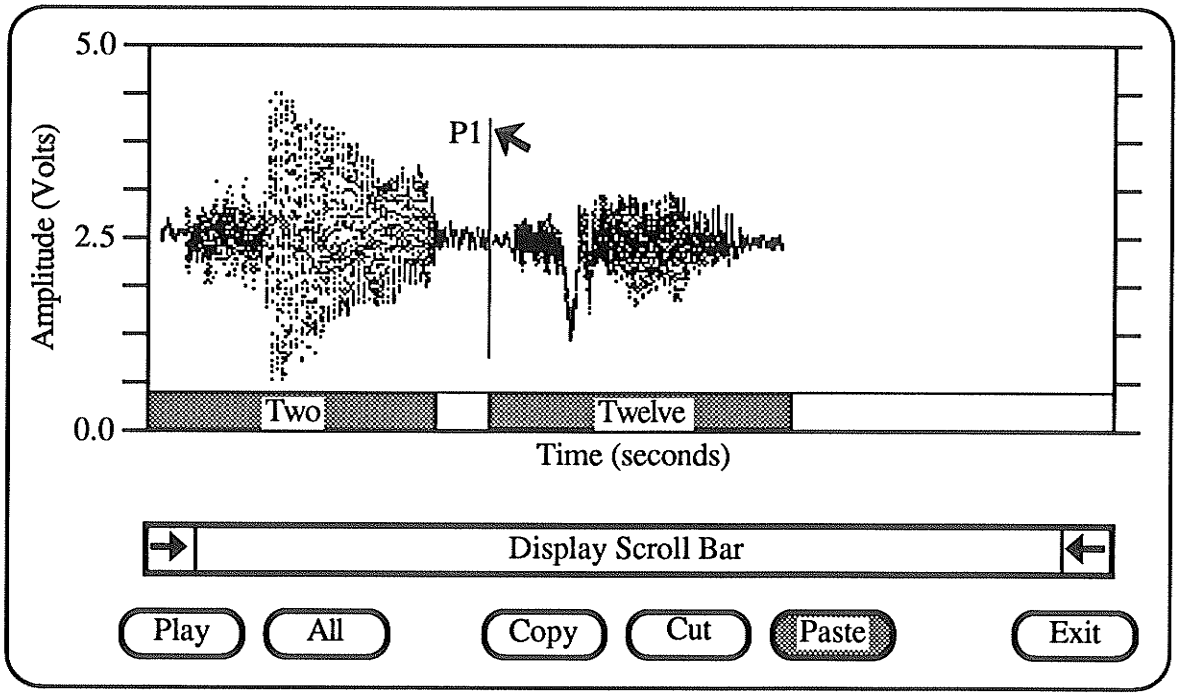


Fig 2.5c Procedure for pasting the waveform of the word 'twelve'. The waveform is pasted near the end of the waveform 'two' (more precisely, at location pointed to by P1) by selecting the paste function of icon, Paste.

The cut, copy, and paste method does not take into consideration the differences in boundary properties that may exist between adjacent words of a phrase and between adjacent phonemes of individual words. However, because human speech perception plays a significant role in understanding speech, the boundary conditions may not adversely affect intelligibility. Whether or not the boundary conditions affect intelligibility depends on how 'bad' amplitude or frequency is mismatched.

2.2.4.2 Amplitude Interpolation

Proper speech splicing entails some consideration of the boundary conditions. One way of matching the boundary amplitudes of the phonemes to be spliced is by amplitude interpolation. This method compares the amplitude envelopes of the phonemes to be spliced. Starting at some point in the phoneme (near the beginning, middle, or end), the

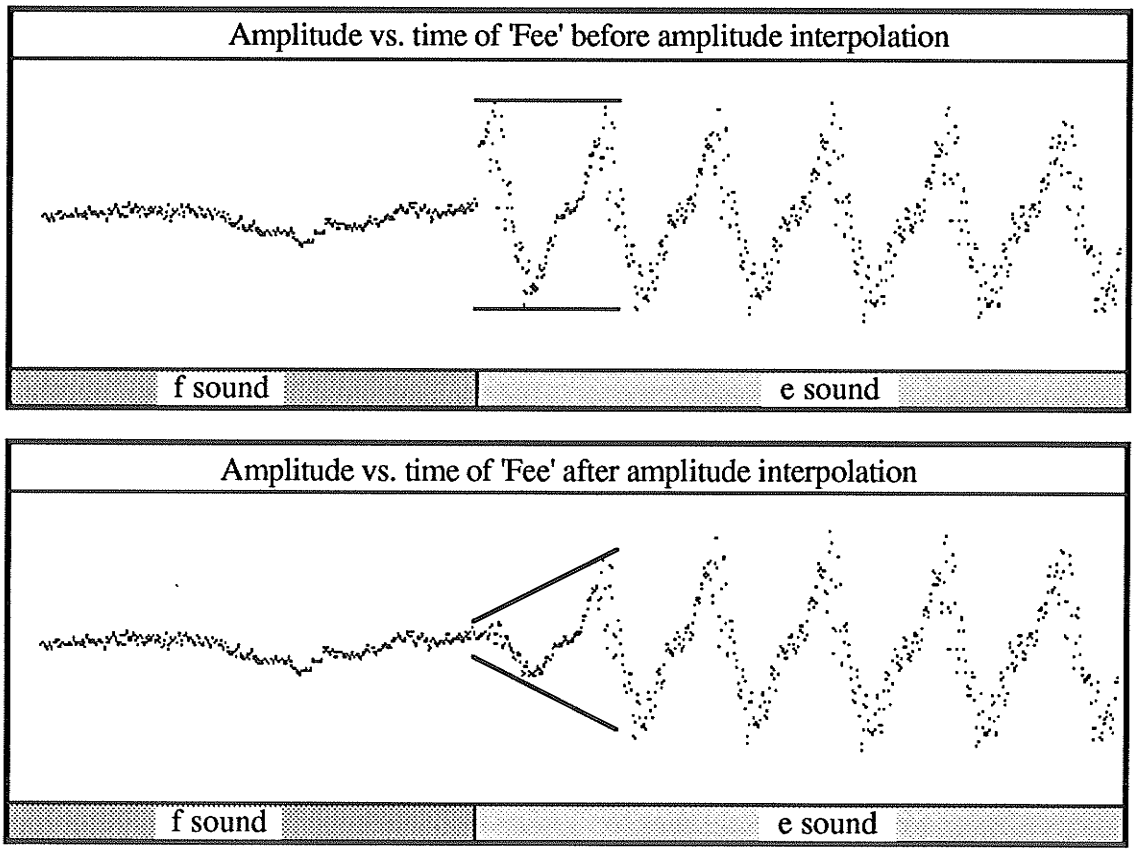


Fig. 2.6 'f' spliced with 'e' using amplitude interpolation.

amplitude envelope of one or both phonemes is increasingly scaled towards the boundary until the envelope amplitudes match. Scaling may be done linearly or non-linearly. Figure 2.6 shows an example. The selected 'f' sound in 'fit' is put together with the selected 'ee' sound in 'beet'. As can be seen in the top waveform, the amplitude envelopes are mismatched. The interpolator matches the amplitude envelopes at the boundary by scaling the envelope of the 'ee' sound.

As with the copy, cut, and paste method, a disadvantage is the amplitude interpolator entails some repetition. The user must repeat the procedure until the right combination of parameters, i.e., linear or non linear scaling, scaling factor, and boundary depth, is chosen. Furthermore, the judgment is subjective and may vary from one person to the next. Another disadvantage is the amplitude interpolator does not take into consideration differences in duration of one or both phonemes to be spliced. The discontinuity at the boundary may be because one phoneme does not continue long enough in order to meet the adjoining phoneme. Another disadvantage is the amplitude interpolator directly modifies the waveform. There are some instances where the integrity of the original waveform is to be left intact, while some other method achieves boundary match and continuity.

2.2.4.3 Linear Predictive Extrapolation (LPE)

The waveform extrapolator can achieve continuity and match at the boundary, while preserving the character of the original phonemes. The waveform extrapolator may achieve a match between two phonemes by introducing an inherently characteristic binding segment at the boundary. As its name suggests, the waveform extrapolator predicts future samples of the waveform based on past samples. This is appropriate for the left hand side phoneme to be spliced. However, for the right hand side phoneme, the extrapolator postdicts or estimates preceding samples based on present samples. Figure 2.7 shows a model for

concatenating phonemes using linear predictive extrapolation. The waveform extrapolator consists of the following three steps: i) linear prediction, 2) linear postdiction, and 3) averaging.

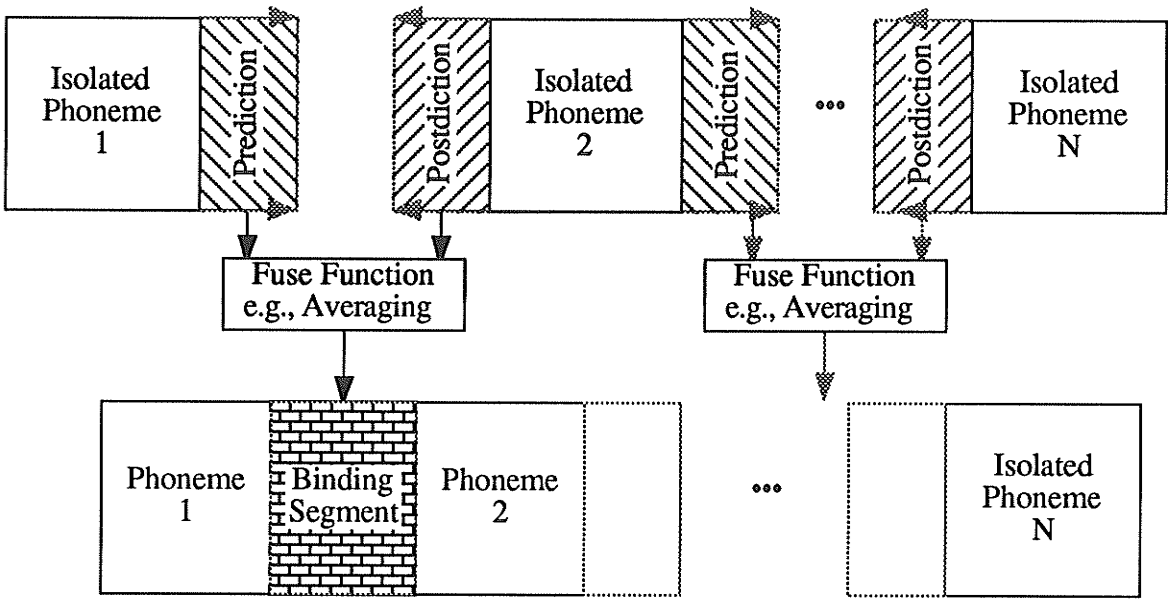


Fig. 2.7 Model of phoneme concatenation using linear predictive extrapolation (LPE).

The motivation for the predictive extrapolator is based on the fact that when humans vocalize words, the adjacent phonemes interact and influence their neighbors. It is proposed that linear prediction may model this interaction, because linear prediction has already been shown to provide a sub-model of speech production [Pars86]. In order to take into consideration both adjacent phonemes, however, postdiction and averaging is included. Averaging gives equal weight to both phonemes for the binding segment contribution. However, it may be that one phoneme should have more influence on the other. In this case, a generalized fuse function may be employed [FeKi91].

Figure 2.8 shows an example of postdiction and prediction of an actual waveform for the phoneme /IY/, as in 'feet'. The objective is to make the phoneme /IY/, spoken in isolation, sound longer and prepare it for eventual splicing using the binding segment

method. The first step consists of linear prediction. From Eq. 2.1, one hundred samples (p = 100) of the previously recorded phoneme /IY/ are used to predict the next twenty samples. These twenty samples are pasted to the end of the original phoneme using the *Paste* function of the system described above. The next step consists of linear postdiction.

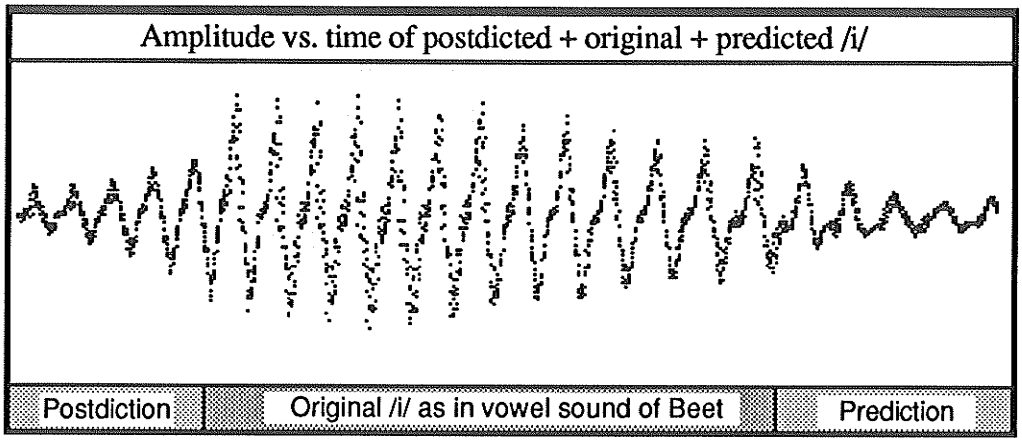


Fig. 2.8 Linear prediction and postdiction of the phoneme /IY/.

From Eq. 2.3, one hundred samples (p = 100) of the previously recorded phoneme /IY/ are used to estimate the previous twenty samples. These twenty samples are pasted at the beginning of the original phoneme using, once again, the *Paste* function of the system described above.

Having described theory of speech synthesis, in particular, the compression technique of adaptive differential pulse code modulation (ADPCM) and three waveform concatenation techniques, namely, copy, cut, and paste; amplitude interpolation; and linear predictive extrapolation (LPE), this chapter now turns to discussing how these concepts can be applied to automated vocal shaping systems.

2.3 Review of Vocal Shaping

This section provides some background information on vocal shaping and automated vocal shaping systems. This review is not comprehensive, but it does cover the pertinent aspects, and it is intended to provide motivation for this thesis and to show how speech splicing may be used in a vocal shaping environment. A definition of shaping and, in particular, vocal shaping is discussed. Following this is a description of the structure of automated vocal shaping systems. This description shows the part played by the speech splicing system of this thesis. Finally, the procedure and method of an automated vocal shaping system is discussed.

2.3.1 Vocal Shaping Definition

Shaping is a behavioral modification procedure of a branch of Psychology dedicated to the study and practice of modifying, developing, and training new behaviors in animals and humans. The shaping procedure is modelled after operant conditioning, i.e., the frequency of occurrence of a behavior is increased by making the presentation of reinforcement contingent on the occurrence of that behavior. Shaping involves a series of applications of operant conditioning for smaller changes in behavior because the probability of exhibiting a complex behavior without prior and similar experience is low. Furthermore, shaping is based on the realization that complex acts are piece-wise continuous in nature [Skin53]. Most complex acts consist of a sequence of smaller actions, which, when serially enacted, form the behavior. An example of this is the method by which circus animals are trained to perform 'tricks'. For instance, in order to get a lion to sit up on its hind legs on top of a table, the trainer would break down this complex behavior into component parts. The lion is trained to perform each part individually and sequentially, starting with the first small act, which may be to get the lion to just look at a table. Each occurrence of the

component part is reinforced, most probably in this case by dispensing food to the lion. After a component part is mastered, reinforcement of that behavior is halted (extinguished), and the next behavior is attempted. This process of reinforcement followed by extinction is continued until the lion exhibits the total complex act. Shaping can be defined as a behavioral modification process consisting of the successive reinforcement of closer approximations and the extinction of previous approximations until the target behavior is achieved [MaPe88].

Vocal shaping is dedicated to the study and practice of modifying vocal behavior of speech disabled individuals [PeKR87]. The process of vocal shaping involves four parameters: (i) target behavior, i.e., correct or desired pronunciation of an utterance; (ii) starting behavior, i.e., a student's initial vocalization of an attempted utterance; (iii) advancement and regression criteria; and (iv) step size [Desr90]. Flexibility in the system is allowed in that the advancement and regression criteria and the step size are derived from a given target response and starting behavior. In this way, the system can adapt itself to individual differences.

The conventional method of administering vocal modification is a long, tedious, and subjective process, because it involves direct interaction between speech therapist and student. After each incremental response, the teacher must assess the behavior, decide whether it is advancing or regressing, and alter the step size (if required). This assessment can be very subjective in nature, and the decisions made by the same therapist can vary from one time to another as well as between different therapists. "Automated systems may aid in this process by increasing the precision by which assessment and training procedures may be administered" [Desr90].

2.3.2 Automated System Architecture

Current research [Love91] proposes a phonemic recognition system for automated vocal shaping. This system consists of three phases, i) acquisition of training data, ii) configuration and training of recognition system, and iii) phoneme shaping. The system is run on a Macintosh IIsi computer and provides a very nice user interface to facilitate easy acquisition, configuration, and shaping.

2.3.2.1 Acquisition of Target and Training Data

The acquisition phase consists of building a library of test patterns. Target phonemes spoken by a speech therapist are recorded and features are extracted and loaded into the library. Each target phoneme consists of numerous versions, which are associated with labels as follows: excellent, good, fair, poor, and unsatisfactory. The purpose of this library is to provide the student with model pronunciations, to provide distance comparisons, and to configure the recognizer.

2.3.2.2 Configuration and Training of Recognition System

The configuration phase configures and trains the recognition system with the test patterns acquired in the acquisition phase. The recognition system is implemented using an artificial neural network. The purpose of the neural network is to recognize the version of the test pattern the student is attempting to imitate and, also, to provide distance measures.

2.3.2.3 Phoneme Shaping

Once target and training data have been acquired and the recognition system has

been trained, the process of shaping can begin. To start the process, the therapist chooses a phoneme from the library. The computer synthesizes this phoneme and plays it back for the student. The student attempts to imitate the sound, and the computer records the response. The recognition system determines the quality of the response by generalizing and mapping the response to one of its known identities. Distance measures are stored in anticipation of the next response. Based on advancement and regression criteria, the system determines whether reinforcement should be administered. Once a particular version has been achieved, reinforcement for that version stops, and the next best version is attempted. In this way, the system attempts to shape the response of the student through successive versions of the target until the correct pronunciation of the phoneme is vocalized.

2.3.2.4 Word Shaping

Once a set of phonemes has been shaped, the next logical step is to try larger content speech, such as words or phrases. If these words already exist in a library, then speech splicing may be used in the process of shaping these words. The objective is to extract phonemes and or syllables and process them using waveform concatenation techniques in order that they, themselves, may be used for vocal shaping. The idea is that if a student is to learn to vocalize an entire word, it would be easier to proceed in steps by individually shaping the component parts of the word. Once the component parts are learned, then they may be put together more easily in order to form the word.

There are two advantages to this method. The variability introduced by having a therapist utter the phoneme or syllable many times in succession is eliminated by the automated system which is capable of synthesizing the component part exactly the same way every time. Also a therapist may not vocalize the phoneme or syllable exactly as it is vocalized in the context of the word. This is not a problem for a system that extracts and plays back the actual waveform of the phoneme or syllable from the waveform which has

embedded in it the contextual information.

However, if these larger words do not exist in the library, then they would have to be added. There are at least three methods of vocabulary expansion. The first method requires a target acquisition phase, which would record the additional words or phrases spoken by a therapist. A second method, which does not involve a target acquisition phase, involves adding previously recorded phonemes to previously recorded words, removing phonemes or other small units of speech from previously recorded words, or replacing phonemes of previously recorded words. This may be done using the waveform concatenation methods described in Section 2.2. For example, if the words 'dog' and 'add' were previously recorded utterances of the therapist, then the new word 'dad' may be formed by copying the phoneme /D/ from 'dog' and pasting it to the starting of the word 'add' to form the new word 'dad'. A third method of vocabulary expansion is similar to the second method. However, a significant difference is that the source of phonemes to be spliced is the student's vocalizations, rather than the therapist's. The idea is to first shape a specific set of phonemes. From this set new words may be formed by the waveform concatenation methods. Because these new words are formed from the student's own pronunciation of the component parts, the resulting pronunciation of the new words would characteristically and inherently sound as the student's own voice. This may lead to intuitive, natural, and easier shaping.

The second and third methods of vocabulary expansion have the additional advantage of providing experimental continuity. During the long course of shaping the vocal responses of a certain student, that student may become dependent on the specific pronunciation of a particular therapist. If for any reason that therapist is no longer able to continue shaping the student, past accomplishments may be set back and the rate of learning may be decreased. Because these concatenation methods may produce familiar sounding

words that may sound inherently characteristic of the therapist's or the student's pronunciation, the experimental variability of changing teachers during the course of training is eliminated.

2.4 Summary

This chapter provides a review of speech synthesis and vocal shaping. The first part of this review focuses on the techniques of ADPCM, LPC, and waveform concatenation. ADPCM is a data compression technique that is used for speech data transmission, and it achieves a compression ratio of 2:1, while maintaining toll quality. LPC is a speech analysis and synthesis method that is used in this thesis in conjunction with waveform concatenation. Waveform concatenation is a method of synthesizing speech from waveform representations. In particular, new words or phrases are formed by joining waveforms of smaller units of speech. More specifically, new words may be formed by concatenating phonemes or by replacing phonemes of existing words. There are three methods of waveform concatenation: (i) copy, cut, and paste, (ii) amplitude interpolation, (iii) and linear predictive extrapolation. These three methods are used in Chapter VI Speech Splicing Experiments.

The second part of this review shows where and how speech synthesis fits in with vocal shaping. Vocal shaping is a psychological procedure for modifying vocal behavior. Vocal shaping involves the administration of reinforcement for closer approximations and extinction of previous approximations until the target response is achieved. The idea is to shape vocal responses of students by getting them to emulate a succession of target responses spoken by a therapist. Speech synthesis, in particular, speech splicing, may be used to form new target responses. New words or phrases may be formed by adding, removing, or replacing phonemes within existing words. This method is called vocabulary expansion and is useful for providing long term continuity and reducing experimental

variability. In the case of concatenating phonemes spoken by students, this may lead to intuitive and easier shaping, because the new words would inherently sound like the student's own pronunciation.

This chapter deals with the psychological and theoretical aspects of this thesis. The next step is the technical aspect which provides the technical means by which the psychological and theoretical aspects are implemented. The technical description begins with defining the requirements and architecture of a speech processing system, which incorporates the tools of speech splicing. This is followed by the organization, i.e., the technology used in order to implement the architecture.

CHAPTER III
SYSTEM REQUIREMENTS AND ARCHITECTURE

Chapter II provides a review of speech synthesis and vocal shaping, and it describes how speech synthesis, in particular, waveform concatenation, may be used in a vocal shaping environment. Chapter II can be thought of as describing the theoretical and psychological aspects of this thesis. The next step is the technical aspect, which concerns the physical realization of the speech splicing system. This step involves an architectural description (described in this chapter) and an organizational realization (Chapter IV).

This chapter discusses requirements and describes the architecture of a system capable of splicing speech using the waveform concatenation method. In a vocal shaping environment, the entire system would consist of three phases: (i) utterance acquisition for vocabulary expansion, (ii) synthesis by isolated phoneme and/or extracted sub-word concatenation, and (iii) synthesized word shaping. However, this thesis concentrates on the speech splicing part of the system, which consists of phases (i) and (ii). As described in Chapter II, Section 2.3.2.1, the utterance acquisition phase consists of acquiring *model phonemes* and/or *model words* spoken by a therapist or *shaped phonemes* spoken by a student. Once the utterances have been acquired, the next phase consists of splicing or synthesizing new words using the waveform methods described in Section 2.2.4, namely, cut, copy and paste; amplitude interpolation; and linear predictive extrapolation.

3.1 System Objectives

In order to satisfy the above two major goals of the speech splicing system, certain objectives can be stated as follows:

(a) Real time recording and playing of speech.

(b) Recording and playing time limited only by the available space on the PC's hard disk.

(c) Good quality speech reproduction characterized by at least telephone (toll) quality, while maintaining relatively low transmission rate, bandwidth, and computer memory.

(d) Isolation of computer from other peripheral speech processing hardware.

(e) Portability and compatibility of peripheral speech processing hardware and host computer.

(f) Intuitive and easy to use interface of host computer.

(g) Main host software including library formulation; data compression and decompression; and amplitude versus time plot of speech waveform allowing the user to display, select, and playback any portion of a digitally recorded speech waveform.

(h) Additional waveform synthesis software including intuitive copy, cut, and paste; amplitude interpolation; and linear predictive extrapolation;

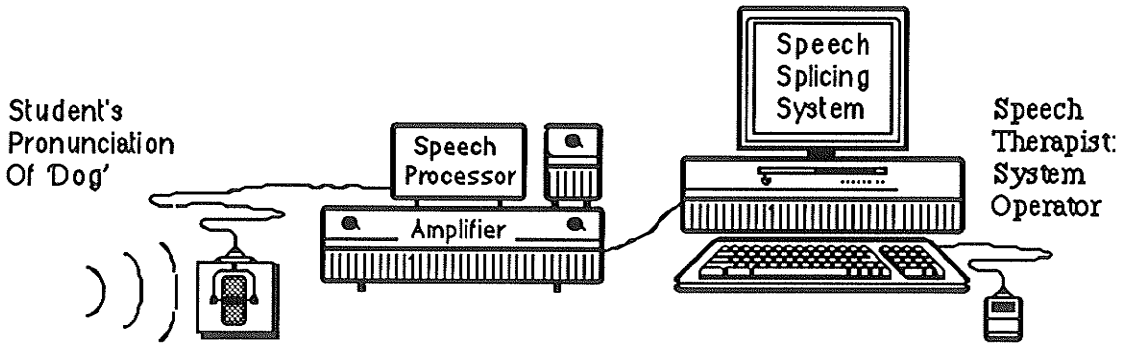


Fig. 3.1 External view of CASS system.

3.2 System Structure

Figure 3.1 shows an external view of the entire vocal shaping system. This figure

shows the word shaping mode of operation. It is assumed that the targets of three previously shaped phonemes (/D/, /AA/, and /G/) have been stored in a library and subsequently spliced to form the word 'dog'. To get a better idea of how this system may work , let us consider an internal view, as shown in Fig. 3.2. As shown in this figure, the system consists of a speech processor, memory manager, serial communications channel, and host computer.

3.2.1 Speech Processor

The speech processor is responsible for recording and playing speech. The sequence of events transpiring during record and playback mode can be described as follows: During recording, the speech processor is responsible for inputting the analog representation of the speech waveform, converting the waveform's analog representation to a digital representation, compressing the digital representation, and transmitting the compressed speech to the next unit, the memory manager. During playback, the speech processor is responsible for capturing the compressed representation of the synthesized word from the memory manager, decompressing it to a linear digital form, converting from a linear digital form to an analog representation, and outputting the analog representation of the synthesized word to an amplifier and eventual speaker for the benefit of the listener.

The inputting and outputting functions involve adhering to certain aspects of signal theory. Inputting must pre-process the speech signal. Pre-processing involves filtering the speech signal. Filtering is required in order to remove every other signal present in the source emanating from the microphone, except the speech of interest. This includes ambient and power source (60 Hz) noise. Electrical signal filtering can be thought of as attenuating (ideally to zero) the component frequencies of the noise to be removed, while passing (usually with unity gain) the signal of interest. Removal of noise present in a speech signal is usually done by inputting the microphone signal through a bandpass filter

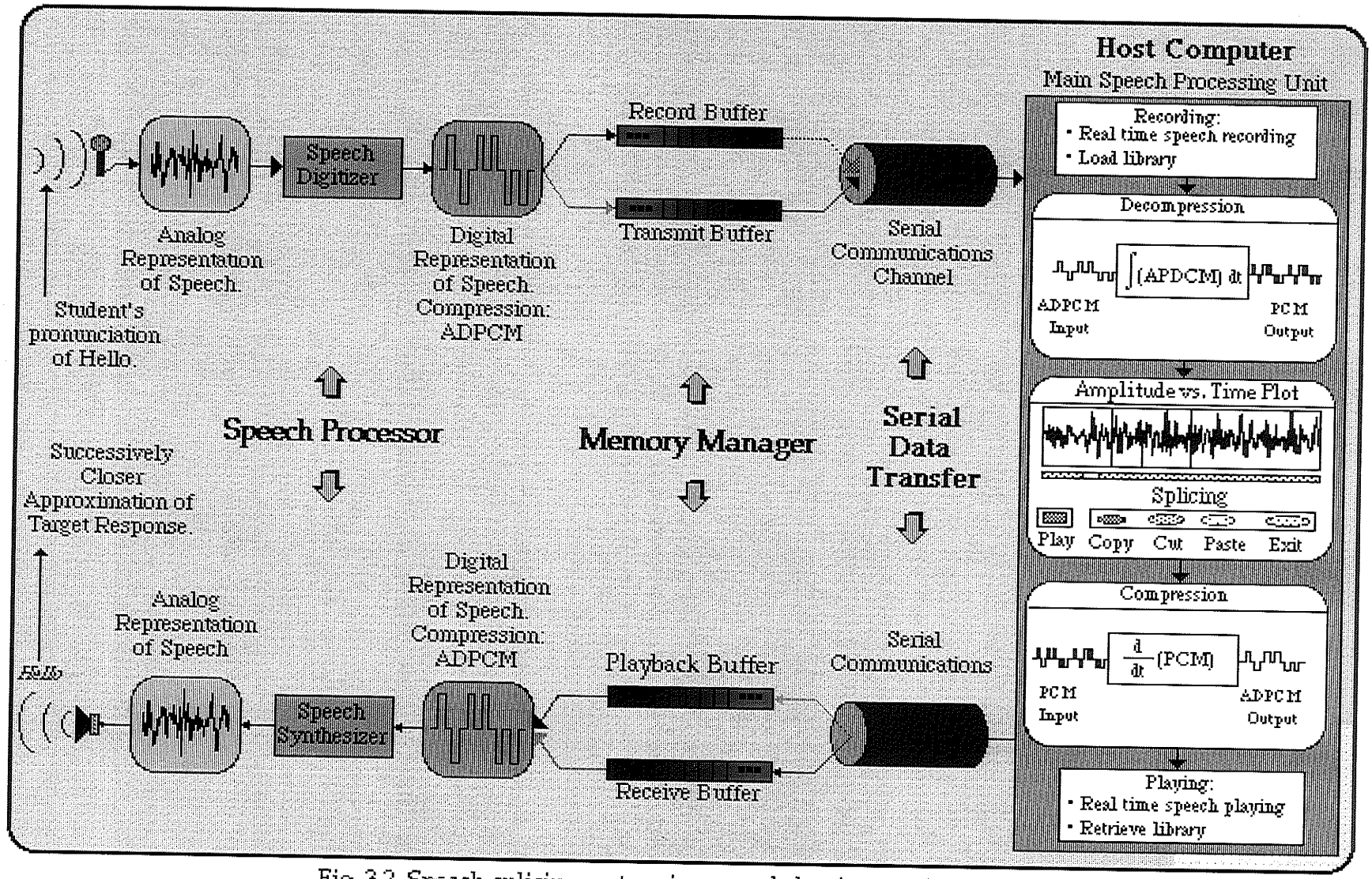


Fig. 3.2 Speech splicing system in a vocal shaping environment.

of bandwidth of approximately 3000 Hz, with cutoff frequencies located at 300 Hz and 3300 Hz. Signals consisting of frequencies outside the band 300 to 3300 Hz are attenuated, while signals within the band are passed unmodified. This bandwidth is sufficient for speech because the essence of speech is contained within 300 to 3300 Hz [Klim87].

Outputting must post-process the speech signal. Post-processing also involves filtering. In this case filtering must be done in order to remove high frequency components introduced by the digitization process. More detail on post-processing is given in Chapter IV.

Another function of the speech processor is to provide transformations or conversions from the analog to the digital domain and vice versa, ADC and DAC, respectively. There are certain advantages of using digital technology. The digital domain facilitates further processing of speech. Speech is much easier to modify in the digital domain. Once in the digital domain, any portion of the speech waveform can be visually associated with its sound. This facilitates copying, cutting, and pasting functions. Furthermore, the accuracy with which speech may be modified is determined by the sampling rate. For example, if speech is sampled at 8 kHz, then the resolution of any copy, cut, or paste is 125 μsec. Another advantage of recording and storing speech in digital form is that it may never degrade.

One of the disadvantages of using digital technology is the cost of the resulting transmission rate, bandwidth, and computer memory. As discussed in Chapter II, a method of reducing these costs is data compression. A suitable compression technique for the application of interest is adaptive differential pulse code modulation (ADPCM). ADPCM compression is used mainly because of its ability to reduce the above costs by one half, while maintaining toll quality.

3.2.2 Memory Manager

The memory manager is a buffer that controls the communication of speech data between the speech processor and host computer. The memory manager provides a continuous flow of data between these two systems which have different data transmission requirements. This buffer is included in the system design in order to realize real-time recording and playing of speech, with recording or playing time limited only by the available space on the hard disk.

If the buffer is omitted from the design, there are two reasons preventing real-time recording and playing of speech. Even though ADPCM is capable of reducing costs by one half, the resulting transmission rate is still quite high and demanding. For example, if speech is sampled at 8 kHz, then the ADPCM transmission rate is 4 kHz. If a 4-bit quantizer is used, this means that a byte of ADPCM speech data is transmitted every 250 μsec. If speech is to be recorded in real time, then each byte must be transferred to a storage medium, and this transfer must not take longer than 250 μsec, else data will be lost. If a host computer is to be solely responsible for reading and saving speech data, this time restriction would be too demanding. For example, an average access time of a hard disk storage medium is about 18 msec. This access time includes powering up the device and positioning the write or read head.

Computers are much more efficient at processing large blocks of data, rather than one byte at a time. Rather than writing each byte to disk as it is received, the host computer is more efficient at reading a block of data, say 8K bytes, and saving this block to disk. The host computer may do this faster than the next block becomes ready to be read and saved. For example, the host computer would have extra time, t_e, given by Eq. 3.1.

$$t_e = \text{BlockFormT} - \text{BlockRecT} - \text{BlockSaveT} - \text{FPT}$$
$$= (8\text{K bytes})(250\ \mu\text{sec/byte}) - (18\ \text{msec} + \text{block write time}) - \text{FPT}$$
$$= 2.03\ \text{sec} - \text{block write time} - \text{FPT} \qquad (3.1)$$

BlockFormT is the time required by an external device to form an 8K block of data. During this time, the host computer is allowed to spend *BlockRecT* time receiving the previously formed block, *BlockSaveT* time saving the current block to disk, and foreground processing time (FPT) doing system tasks. *BlockSaveT* time includes disk access (about 18 msec) and the time required to write 8K bytes to disk. Foreground processing time is the time required by the host computer to perform system tasks, such as maintaining communication at the user interface and refreshing Dynamic Random Access Memory (DRAM). These tasks are considered high priority, particularly the DRAM refresh cycle. Reading speech data and storing same to hard disk is considered a lower priority background task.

However, because of the variability in the time required to write to disk and to perform foreground processing, whether or not the host can complete these tasks before the next block is ready for transmission is determined experimentally. For example, the time to write 8K bytes to disk varies because of disk segmentation. Because files are constantly being written, modified, and deleted, and because the operating system is forced to make efficient use of the disk, there is a good chance that an 8K block of data will not be written contiguously on disk. As a result, the time taken to write to disk is much longer than expected, where the expected time is the time required by a direct memory access (DMA) controller to write a block of RAM to disk.

Because of the demanding transmission rate and high priority foreground tasks, the system may have problems recording or playing speech in real time. In order to allow the

host computer the time it requires to do system tasks in the foreground and speech processor requests in the background, some sort of buffer is required. This buffer must be able to take over the task of communicating speech data with the speech processor, while communicating blocks of speech data with the host computer.

There are several buffer concepts which satisfy the requirements of real time recording and playing of speech. This thesis presents designs of two buffers. The buffer implemented in the system is called a dual pointer *First In First Out (FIFO) buffer*, implemented in software. This buffer is described in Chapter IV, Section 4.2. An alternative buffer, called a *swinging buffer*, implemented in hardware is discussed in Chapter V.

3.2.3 Serial Communications Channel

The serial communications channel is the medium through which speech data is transmitted between the host computer and memory manager. It is included in the system design in order to achieve isolation and portability objectives.

Directly connecting the speech processor to the host computer may not be a good idea. Some kind of trouble, say an electrical problem, happening with the memory manager or speech processor may damage host circuitry if there were a direct connection. The serial communication channel isolates the two systems fairly well.

Most host computers have serial port interfaces which are compatible with the RS-232C standard. Therefore, by using a RS-232C compatible serial communications channel, this system may work on other compatible host computers. The portability objective also depends on the software and operating system used by the host computer.

3.2.4 Host Computer

The host computer must play a central and foremost role in the speech splicing system. For the purpose of speech splicing research, it really does not matter what brand of host computer is used, so long as it is capable of the objectives. However, because of technological and physical availability, a decision as to which which type of host computer to use must be made.

3.2.4.1 Choosing a Host

There are at least two computers, Macintosh and IBM, capable of performing the task of a host. The Macintosh is probably the best choice because sound I/O technology is much more advanced. In particular, a Macintosh IIsi has built in sound input and output, capable of monaural 8-bit voice input, via an included electret microphone, and stereo output via a minijack output located in the back. The new ROM based sound manager of System 7.0 offers very easy and flexible voice recording and playing, with selectable sampling frequencies of 11 and 22 kHz and selectable compression techniques. However, recording speech to disk is not yet possible with the sound manager. Record and playback time is limited to allocated RAM. With extra software, such as MacRecorder and SoundEdit, it is very easy to edit speech. In particular, the task of cutting, copying, and pasting is very easy and graphically oriented [MacU90]. This facilitates speech splicing.

The Macintosh IIsi, together with available speech editing software, achieves most of the objectives of the speech splicing system. Perhaps the only additions to the system would be a speech splicing system application program, which interfaces the user to phoneme acquisition, splicing, and word or phrase reproduction. This amounts to saying a speech splicing system is available on the Macintosh.

Unlike the Macintosh, voice input and output is not directly possible with an IBM or compatible. The IBM has a built in amplifier and speaker, but its sound reproduction is limited to simple beeps or tones. If the IBM is to be used as host computer, then external circuitry, such as the speech processor, memory manager, and serial communications channel as described above, would be required. This thesis uses the IBM as host because currently there are more IBMs in the market, designing with the IBM is more challenging, and a similar speech splicing system on the IBM is not currently available.

3.2.4.2 Requirements

The host computer must provide a user interface to the system. Most of the other system components, i.e., serial communications channel, memory manager, and speech processor, should be virtually hidden from the user. When the user requests to record or playback speech, a command should be entered through the interface, and the host computer should take care of the rest of the details. For example, if the user wishes to start recording, the command should be issued through a visual button icon on the video display unit (VDU), analogous to depressing a record button on a tape cassette recorder. Similarly, in order to perform some simple speech editing tasks, the user should be able to access any portion of speech anywhere in the speech data file. This access should be done through a display of an amplitude versus time plot of the speech data file, as indicated in Fig. 3.2.

3.2.4.3 IBM Software System Hierarchy

All operations of the host computer in the speech splicing system are implemented in software. Software on the IBM computer consists of four levels, high level language, DOS calls, BIOS calls, and assembly language. The IBM recommended design philosophy [NoWi85] is as follows: Write the application program using only high level language. If

the application calls for some functions not provided by high level language, use either ROM BIOS or DOS. BIOS is a basic input output system located in ROM. DOS is located on the start up disk and is partially loaded into RAM on system boot-up. BIOS and DOS provide screen, disk, serial communications I/O, and many other low level functions. All BIOS and DOS functions are invoked by software interrupts. Each interrupt is associated with a location in an interrupt vector table. Each location contains an address of the selected routine. As long as the PC is controlled through high level language, DOS, or BIOS the designer is safe from compatibility problems, i.e., there is a good chance that the application program will work on other PCs. If the application calls for some manipulation of hardware or peripheral devices not provided by ROM or BIOS or direct control of the CPU is required, then the last alternative is to use assembly language. However, use of assembly language to manipulate hardware and peripheral devices is not recommended because hardware and peripheral devices vary from one PC model to the next.

3.2.4.4 User Interface

The user interface is superficially the most important part of the system. The 'goodness' of most software is judged by the user interface. Questions such as 'Is the system intuitive and easy to operate?' and 'Is the appearance of the software inviting and eye catching?' are predominantly the ones that must be answered by the software. When designing a user interface, the programmer has two alternatives, which are (i) design the interface from scratch using a high level language, DOS, BIOS, and assembly language or (ii) purchase a graphical user interface (GUI) [CoLa90], which typically contains source code and libraries for creating pop up menus, windows, dialogue boxes, buttons, and switches. This thesis shows how to design a simple interface using the former method. This is described in Chapter IV, Section 4.4.1.

3.3 Summary

This chapter discusses requirements and describes the architecture of the speech splicing system. The system is required to be capable of two major goals: (i) utterance acquisition for vocabulary expansion and (ii) synthesis by isolated phoneme concatenation and synthesis by extracted sub-word concatenation. In order to satisfy these requirements, the architecture of the system must consist of a speech processor, memory manager, a serial communications channel, and a host computer. Under this architecture, the system is capable of real-time recording and playing of speech, disk capture, ADPCM compression and decompression, amplitude versus time plot of speech waveform allowing the user to display, select, and playback any portion of a digitally recorded speech waveform, and additional waveform synthesis software including intuitive copy, cut, and paste; amplitude interpolation; and linear predictive extrapolation.

Having described the architectural requirements, the next chapter shows what technology is used in order to implement the architecture. Chapter V provides every detail of the implementation required in order the system be reproduced.

CHAPTER IV
DETAILED SYSTEM DESCRIPTION

This chapter describes the organization of the speech splicing system, i.e., the technological implementation. The system's main functions include recording and playing speech using the adaptive differential pulse code modulation (ADPCM) compression scheme [OkiS90], acquiring and transmitting digital speech samples to and from target and host computer subsystems using serial transmission, storing digital speech samples in a host computer, and, finally, processing of the stored digital speech. The system center, an IBM compatible host computer, is menu driven with a graphical user interface. The system is capable of real-time recording and playing of speech, for lengths of time limited only by available space on the hard disk. The system includes code conversion routines for compressing and decompressing pulse code modulation (PCM) and ADPCM formatted speech, respectively. The system is designed specifically to facilitate easy and user friendly speech editing tasks, such as cutting, copying, pasting, and splicing. To this end, the time domain plotting feature, with frame by frame or sample by sample scrolling, allows the user to view and manipulate any portion of the speech waveform at any point in the file. Finally, the system software also includes routines for implementing the linear predictive extrapolation method of waveform synthesis.

Due to the modular architectural design of the system, experienced software developers may easily add features, such as frequency and spectral plots. Other more advanced features may include special effects, such as echo, reverb, flange, and pitch control. Furthermore, experienced hardware designers may easily incorporate into the design other speech processor integrated circuits (ICs) using different compression schemes, opening the door to a wide variety of digital speech processing.

4.1 Speech Processor: the MSM6258 MPU Interface Version [OkiS90]

The speech processor used in this thesis is the MSM6258VJS microprocessor unit (MPU) interface version, manufactured by Oki Semiconductor. It is a complex and highly integrated speech digitizer and synthesizer featuring the ADPCM method of data compression. It is designed to be interfaced with an 8-bit microprocessor (μP), such as the Motorola 6802 μP. The speech chip includes internal analog to digital and digital to analog circuits, timing and control synchronization signals, and selectable sampling frequencies and ADPCM bit numbers. The chip is implemented in CMOS technology for low power consumption.

As shown in Fig. 4.3, the chip internally consists of a command and a status register, an 8-bit analog to Digital Converter (ADC), an ADPCM analysis and synthesis unit, a data I/O buffer, and a 10-bit Digital to Analog Converter (DAC).

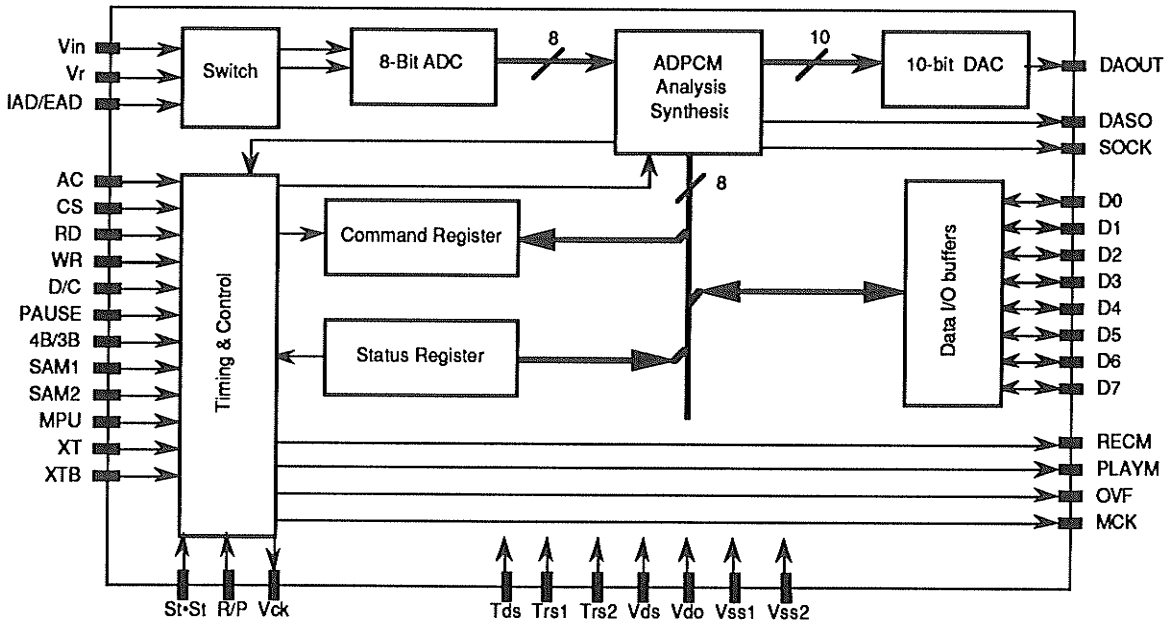


Fig. 4.3 MSM6258 block diagram.

During the record mode, the ADC periodically samples and converts the input analog speech waveform to an 8-bit digital representation. These digital samples are passed

to the ADPCM analysis stage, where each 8-bit digital representation is compressed to either a 3-bit or 4-bit ADPCM representation. Two successive ADPCM representations are concatenated to form an 8-bit data byte, which is loaded into the I/O buffer and, thus, output at pins D0 through D7. Output control signals generated by the Timing and Control unit indicate when an external device may read each 8-bit data byte. During the playback mode, control signals generated by the Timing and Control unit indicate when an external device may write 8-bit data bytes to the I/O buffer via pins D0 through D7. These data bytes must consist of two ADPCM nibbles formatted exactly as that done by the analysis stage. These data bytes are fed into the ADPCM synthesis stage, where each nibble is extracted and decompressed into a 10-bit representation (without introducing any new information). Each 10-bit digital representation is fed into the 10-bit DAC, and the resulting analog signal is output at pin DAOUT.

4.1.1 Functional Pin Description

The MSM6258VJS comes in a 44-pin Plastic Leadless Chip Carrier (PLCC) package. Fig. B1 shows the top view and the pin diagram of the chip. The 44 pins can be grouped into three categories, voice Input/Output (I/O), MPU interface, and miscellaneous.

4.1.1.1 Voice Input/Output (I/O)

The pins associated with voice I/O include VI, VR, SAM1 and SAM2, and DAOUT.

VI: The analog Voltage Input (VI) of the speech waveform is input through pin 38. This signal must be pre-amplified and low pass filtered. The circuit used to perform the pre-amplification and the filtering is shown in Fig. 4.4.

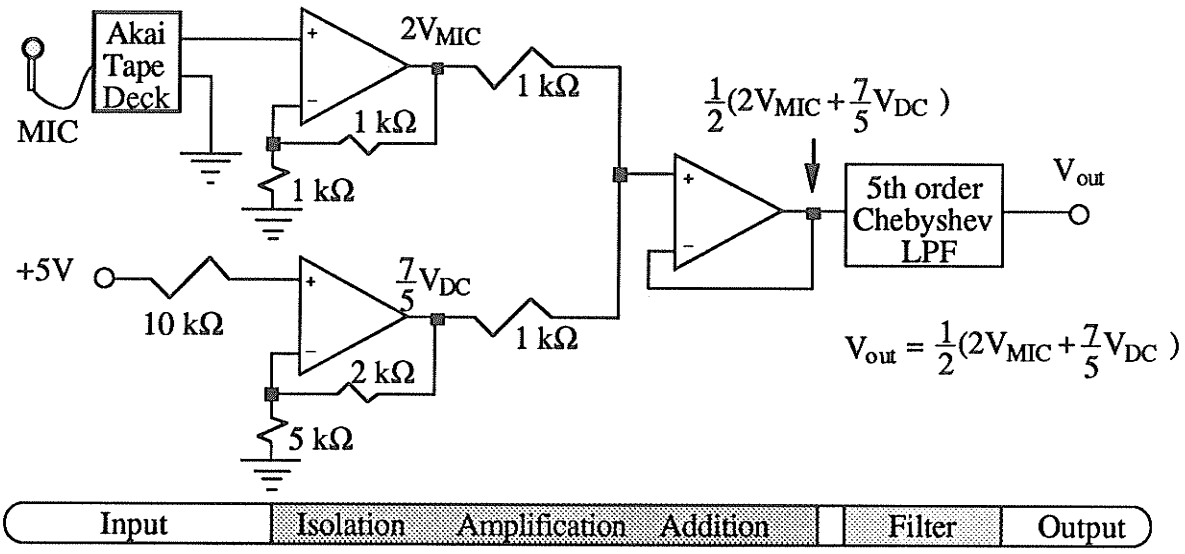


Fig. 4.4 Pre-processing circuit.

The input voltage at pin VI (V_{out} in Fig. 4.4) must be greater than 0V and less than 5V, since the reference voltage used by the internal ADC is 5V. If VI exceeds 5V or falls below 0V, clipping will occur and the ADC section of the chip may be permanently damaged.

The signal-to-noise ratio (SNR) is dependent on the input dc bias. In order to determine the dc bias required for optimal SNR, testing of the pre-processing circuit with an input sine wave of frequency 1 kHz is done. It is found that a dc bias of 2.5 to 3V gives the best reproduction, while decreasing the bias from 2 to 0V yields decreasingly poorer quality. Testing is done by observing the reproduction on an oscilloscope to get a rough estimate of the optimal dc bias and then by using SNR measurements to fine tune the bias.

Assuming a sampling frequency of 8 kHz, the frequency content of the input speech signal should be in the bandwidth of 300 to 4000 Hz in order to eliminate the low frequency (e.g., 60 Hz) noise and to prevent aliasing. However, the bottleneck is the frequency response of the speech chip. The frequency response of the chip is considered for two cases. When the input waveform varies between 0 and 2.5V, the bandwidth of the speech

chip is about 3200 Hz and about 2000 Hz when the input varies between 0 and 5V. Assuming the best response, the input waveform may be filtered to pass the frequency band of 300 to 3200 Hz. This is accomplished through the use of a 5^{th} order Chebyshev filter [OkiS90, pp. 378-380].

VR: The Voltage Reference (VR) of the internal ADC is input through pin 37. The reference voltage is nominally the supply voltage, VDD.

SAM1 and SAM2: The SAMpling frequency is determined by the logic levels input to pins 12 and 13, respectively. Table 4.1 shows the possible sampling frequencies as determined by SAM1 and SAM2. For an oscillation frequency of 4.096 MHz, any of the frequencies shown in the table may be selected as the sampling frequency.

Table 4.1 Sampling frequency selection.

SAM1	L	H	L	H
SAM2	L	L	H	H
Sampling Frequency(kHz)	4.0	5.3	8.0	Inhibited

DAOUT: During playback, the synthesized speech waveform is output at pin 28, the Digital to Analog OUTput. The DAOUT pin provides a staircase type signal varying between 0 and 5V and centered at 2.5V, as shown in Fig. 4.5. Also, during recording, DAOUT monitors the input waveform at pin VIN, with a slight delay. This monitoring is useful for debugging and is an indication that the ADC and the DAC sections of the speech chip are in working order.

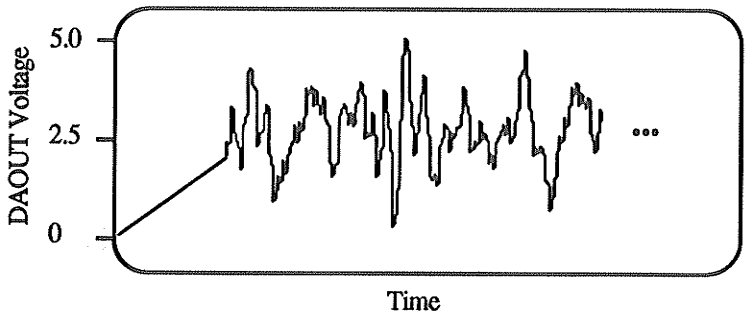


Fig. 4.4 DAOUT voltage vs. time. DAOUT is the output of the DAC and, therefore, it is a stepwise function containing high frequency components.

The stepwise function provided by DAOUT must be filtered in order to remove the high frequency components introduced by the DAC. Although there are many active filter designs, such as the Butterworth, Bessel, and Chebyshev filters, a suitable filter is the 5^{th} order Chebyshev. The Chebyshev filter is capable of achieving sharp attenuation characteristics with a relatively small number of component parts.

After passing the signal from pin DAOUT through the 5^{th} order Chebyshev filter, the stepwise function takes on a smoother look, as shown in Fig. 4.6.

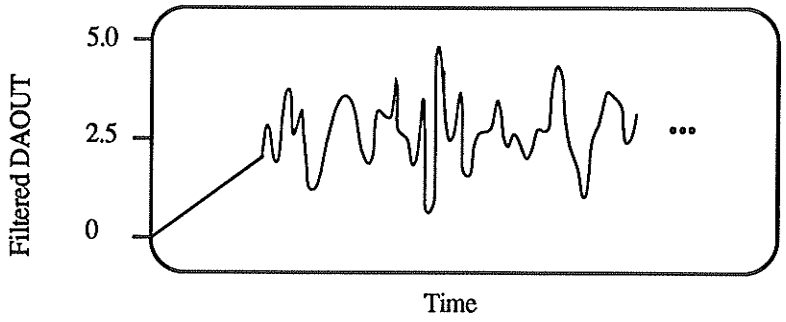


Fig. 4.6 Filtered DAOUT voltage vs. time.

4.1.1.2 MPU Interface

The pins associated with the MPU interface include MPU, CS, VCK, MCK, RD, WR, D/CB, D0-D7, 4B/3B, and test pins. These pin are intended to be used by a microprocessor

or microcontroller for the purpose of control and synchronization.

MPU The MicroProcessor Unit input pin 17 selects the MPU interface mode of the speech chip. (Note: there are two modes of the speech chip, stand-alone and MPU). When a logic 1 is input to MPU, the internal circuitry is set to communicate with an external Central Processing Unit (CPU) of a microprocessor or microcontroller.

CSB A logic 0 placed on input pin 30 Selects and enables the speech Chip to communicate with an external CPU. When this pin is logic 1, the chip is disabled and the data bus is placed on high impedance. In order to prevent bus contention, the external MPU should enable the chip only during the data data access mode, i.e., when writing commands and data or reading status and data to and from the speech chip, respectively.

VCK The Voice sampling ClocK at pin 3 is an output signal indicating the sampling frequency selected by SAM1 and SAM2. When AC is logic 0, the duty cycle of VCK is 50%, and when AC is logic 1, VCK is logic 0.

MCK The Microprocessor ClocK is an output intended to be used by a MPU for synchronization. During record or playback, pin 14 outputs a square wave of frequency one-half the sampling frequency and of, approximately, 20% duty cycle. During record, MCK indicates when data can be read from the speech chip. During playback, MCK indicates when data can be written to the speech chip.

D0-D7 The bi-directional data bus lines located at pins 10, 9 , 8, 7, 2, 1, 43, and 42, respectively, communicate ADPCM coded data and commands and status. During record or playback, the data bus transmits a pair of ADPCM nibbles every MCK. Also, the data bus is intended to be used by an external MPU in order to write commands to the speech chip or to read status information being output by the speech chip.

4B/3B The input pin 22 selects different ADPCM nibble lengths. When 4B/3B is logic 1, the 4-bit ADPCM nibble is chosen, and, when 4B/3B is logic 0, the 3-bit ADPCM nibble is chosen. Table 4.2 shows the ADPCM composition on the data bus. Note that each byte of data contains two different nibbles. When two 3-bit ADPCM nibbles are concatenated to form an 8-bit byte, the Least Significant Bit (LSB) of each nibble is automatically set to logic 0.

Table 4.2 ADPCM bus composition.

Bus Lines	D0	D1	D2	D3	D4	D5	D6	D7
4-Bit ADPCM	B0n	B1n	B2n	B3n	B0n+1	B1n+1	B2n+1	B3n+1
3-Bit ADPCM	00	B0n	B1n	B2n	00	B0n+1	B1n+1	B2n+1

4-Bit ADPCM:
B3 = Sign Bit
B2 = MSB
B1 = 2SB
B0 = LSB

3-Bit ADPCM:
B3 = 00
B2 = Sign Bit
B1 = MSB
B0 = LSB

Sign Bit = 1 means waveform is descending.
Sign Bit = 0 means waveform is ascending.

RDB At the low to high transition of this active low input, pin 31, an external MPU can ReaD data or status information from the speech chip.

WRB At the low to high transition of this active low output, pin 29, an external MPU can WRite data or command information to the speech chip.

D/CB Input pin 32 selects the speech Data mode or the Command/status mode. When D/CB is logic 1, the data bus provides speech data. When D/CB is logic 0, commands may be written to the speech chip or status information may be read.

The pins, TDS,TRS1,TRS2,TSP, TRP, AND TVD are intended for factory testing. These input pins must be set to logic 0 for normal operation.

4.1.1.3 Miscellaneous

The remaining 19 pins are general in nature and include VDD, VSS1, VSS2, XT, XTB, IAD/EADB, PLAYM, RECM, DASO, SOCK, OVF, NC, REC/PLAYB, ST•SP, PAUSE, and VDS.

VDD Pin 35 is the 5V power supply terminal.

VSS1 Pin 11 is the digital ground terminal. All components associated with digital signals and requiring ground signals should use VSS1.

VSS2 Pin 36 is the analog ground terminal. All components associated with analog signals and requiring ground signals should use VSS2.

The input signals REC/PLAYB, ST•SP, PAUSE, and VDS at pins 5, 33, 34 and 40, respectively, are intended for the stand alone version of the MSM6258. For the MPU version their functions are not applicable, and they should all be set to ground level.

XT & XTB The clock circuit can be connected to XT and XTB terminals of the speech chip. XT, pin 15, is an input, and XTB, pin 16, is an output. If an external clock is used, it should be connected to XT, and XTB should be open. In this thesis, the latter method of connecting a clock is used. In particular, the clock of the 6802 μP is connected to XT of the speech processor.

IAD/EADB Pin 41 selects the Internal Analog to Digital converter or an External ADC. A logic 1 on IAD/EADB selects the built in ADC, and a logic 0 on IAD/EADB enables the use of an external ADC. In this thesis, the IAD is used.

PLAYM Pin 26 is the PLAY Monitor. During playback mode this output is logic 1.

RECM Pin 4 is the RECord Monitor. During record mode this output is logic 1.

DASO Pin 24 is used to output serial PCM data to an external DAC. This pin is not applicable when the internal ADC is used.

SOCK Pin 25 is used to clock the serial PCM data being output by DASO. This pin is not applicable when the internal ADC is used.

OVF The OVerFlow output signal at pin 21 gives an indication when the input voice signal exceeds 80% of the dynamic range of the speech chip. It can be used as an input to an Automatic Gain Control (AGC) circuit in order to control the amplitude of the input voice signal.

NC Pins 6, 23, 39, and 44 have No Connection. They may be left open.

4.1.2 Operation

This section describes how an MPU may operate the speech chip. The operation essentially consists of writing speech chip commands, reading speech chip status information, reading speech data during record mode, and writing speech data during playback mode. The MPU must know the structure of the command and status information and the format of the ADPCM data on the data bus. Also, the MPU must know the correct timing at which these data are accessed.

4.1.2.1 Data Bus Control

In order to operate the speech chip effectively, the MPU must control the data bus. The MPU is the master and the speech chip is the slave. The master informs the slave of the operation to be carried out by placing the appropriate logic levels on the input control signals D/CB, CS, RD, and WR. Logic combinations of these control signals have specific meaning to the speech chip. Table 4.3 shows the required logic levels for each of the available operations, which are recording, playing, status output, and command input. Note that the data bus can generally be used in two ways, speech data I/O or command/status I/O.

Table 4.3 Speech chip operation codes.

CS	D/CB	RD	WR	Operation
0	1	0	1	Speech chip outputs ADPCM data (Recording)
0	1	1	0	Speech chip inputs ADPCM data (Playing)
0	0	0	1	Status output
0	0	1	0	Command input
1	x	x	x	High impedance

4.1.2.2 Command Input

The writing of a command to the speech chip is typically the first operation performed. Table 4.4 shows the available commands and the mapping of the codes to the data bus. For example, the number $00000100_{Binary} = 04_{Hex} = 4_{Ten}$ is the code for the record command, and $00000010_{Binary} = 02_{Hex} = 2_{Ten}$ is the code for the playback command. A command is written to the speech chip by placing the command on the data bus and then setting the appropriate logic levels on the input control signals D/CB, CS, RD, and WR.

Table 4.4 Command codes.

Data Bus	D7	D6	D5	D4	D3	D2	D1	D0
Command	0	0	0	0	0	Record	Play	ST•SP

Start or Stop: ST•SP Code = 00000001
Playback: Play Code = 0000001C
Record: Record Code = 00000100

The MPU must also know the correct timing in order to successfully write commands to the speech chip. Figure 4.7 shows the timing of the control signals D/CB, CS, and WR required in order to write a command to the speech chip. As shown in the figure, a command may be written to the speech chip while the chip is selected or enabled. The assertion of D/CB chooses the command input mode. The assertion of WR follows shortly thereafter. Note that at time labelled A, the logic levels of D/CB, CS, and WR are as those shown in Table 4.3 for the command input operation. Also, note that the command code must be on the data bus at the moment in time when D/CB, CS, and WR are all asserted. Finally, it is the low to high transition of WR, while D/CB and CS are logic 0, that actually writes the command to the internal I/O buffer of the speech chip.

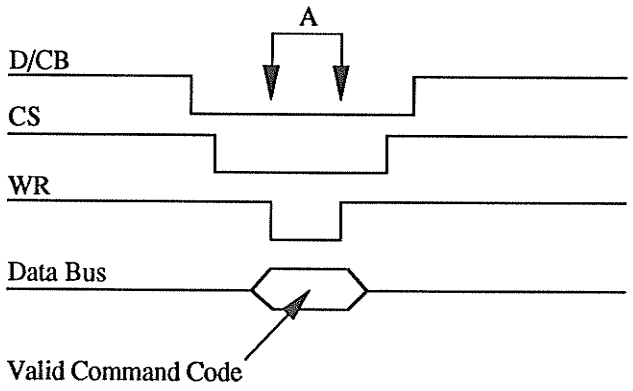


Fig. 4.7 Command write timing.

4.1.2.3 Status Output

The MPU can read status of the speech chip in order to determine what operation is currently being performed. The status of the speech chip may be read by setting the appropriate logic levels on the input control signals D/CB, CS, RD, and WR and then reading the status code off the data bus. The interpretation of the status codes on the data bus is as shown in Table 4.5.

Table 4.5 Status codes.

Data Bus	D7	D6	D5	D4	D3	D2	D1	D0
Status	Rec/Play	x	x	x	x	x	x	x

Playback: Play Status = 1xxxxxxx
Record: Record Status = 0xxxxxxx

The MPU must also know the correct timing in which to read status information off the data bus. Figure 4.8 shows the timing of the control signals D/CB, CS, and RD required in order to successfully read the status of the speech chip. As shown in the figure, status may be read while the chip is selected or enabled. The assertion of D/CB chooses the status output mode. The assertion of RD follows shortly thereafter. Note that at time labelled B, the logic levels of D/CB, CS, and RD are as those shown in Table 4.3 for the status output operation. When D/CB, CS, and RD are logic 0, the speech chip begins placing the code for its status on the data bus. Finally, it is the low to high transition of RD, while D/CB and CS are logic 0, that the MPU actually reads the status off the data bus.

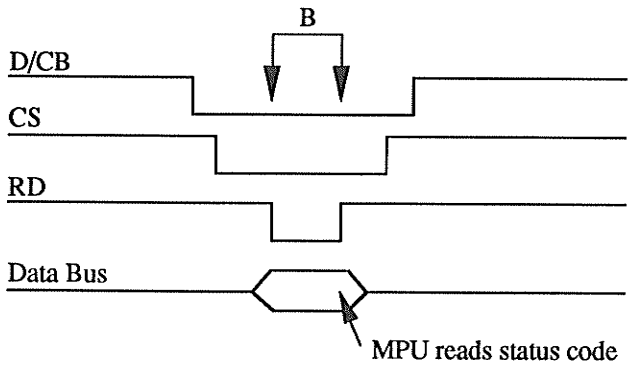


Fig. 4.8 Status read timing.

4.1.2.4 Record

As mentioned earlier, the record mode of operation begins when the MPU writes the record command to the speech chip by using control signals D/CB, CS, and WR along with placing the code on the data bus. In more detail, during recording the ADC digitizes speech samples at the rate of 8 kHz and the ADPCM synthesis unit forms ADPCM bytes at the rate of 4 kHz. In order to inform an external device when each ADPCM byte is available for reading, the speech chip supplies a *data ready* signal, called MCK. This means that recording of speech data is synchronized with MCK. Therefore, in order to read speech data, the MPU synchronizes its read timing signals, D/C, CS, and RD, with MCK. The speech data read timing is shown in Fig. 4.9. The timing is similar to that for reading status, the only temporal difference being that reading status can be done at any time, whereas, reading speech data must occur immediately following the negative edge of MCK. Arrows in the figure show the required causality. In other words, the negative edge of MCK causes the MPU to assert D/CB, followed by CS, and subsequently RD. Note that at time labelled B the states of the control signals D/CB, CS, and RD are as that shown in Table 4.3 for the record operation. A more detailed description of the read timing can be found in Section 5.2.2.1.

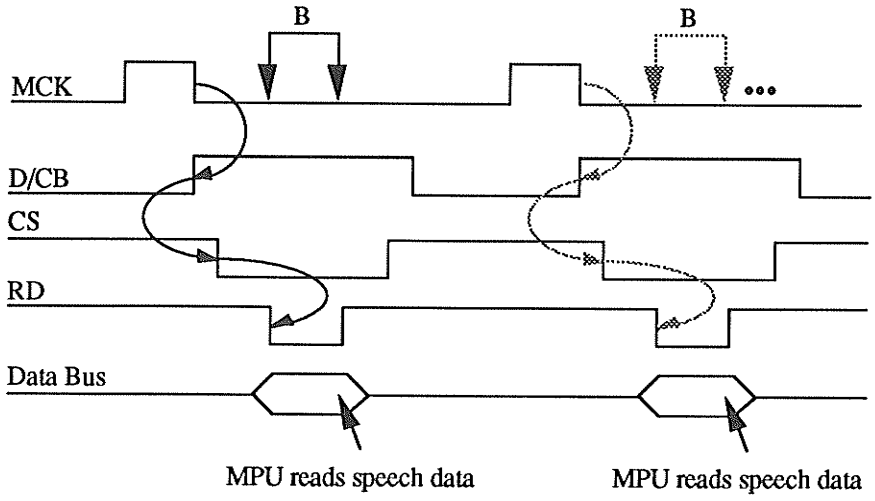


Fig. 4.9 Record timing.

4.1.2.5 Playback

The method for playing speech data is similar to recording. To start the playback mode of operation, the MPU writes the playback command to the speech chip by using control signals D/CB, CS, and WR along with placing the code on the data bus. Once the speech chip latches the command into its command register, playback begins. In more detail, during playback the ADPCM synthesis unit reads ADPCM bytes at the rate of 4 kHz and the DAC outputs an analog sample at the rate of 8 kHz. In order to inform an external device when each ADPCM byte can be written, the speech chip supplies a *ready for data* signal, called MCK. As for recording, playing of speech data is synchronized with MCK. The major difference is that the *ready for data* signal is indicated by the positive edge of MCK. Therefore, in order to play speech data, the MPU synchronizes its play timing signals, D/C, CS, and WR, with the positive edge of MCK, as shown in Fig. 4.10.

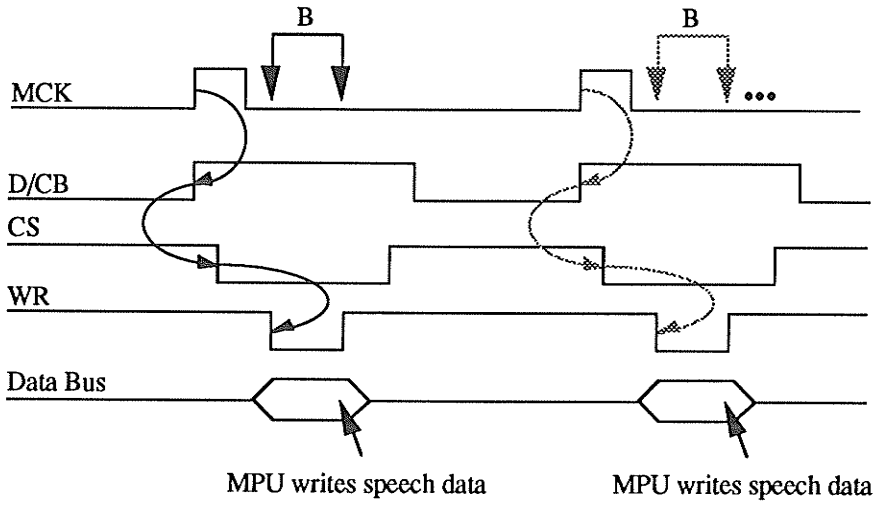


Fig. 4.10 Playback timing.

To stop recording or playing of speech data, the MPU writes a stop command. The method for writing a stop command is similar to writing a record or playback command.

What has been discussed so far in this section is how speech is recorded and played. However, what has not been mentioned is how or where speech data is saved and retrieved in and out of memory during recording and playing, respectively. The MPU version of the MSM6258 does not generate addresses for external Random Access Memory (RAM). It is the responsibility of an external MPU not only to control the data bus, as described above, but also to save speech data during recording and to retrieve same during playing. Ultimately, speech data must be saved in the host computer's private RAM in order to do processing or just to be played back at a later time. The question is the following: How is speech data communicated to and from the host computer?

4.2 Memory Manager: the First In First Out (FIFO) Buffer

This section describes the memory manager. The purpose of the memory manager is to manage the communication of data between the host computer and the speech processor. The memory manager is a memory mapped system, with each device having a distinct address and being controlled by a Central Processing Unit (CPU). As shown in Fig. 4.11, the memory manager consists of two I/O interface ports with one configured in parallel and the other in serial, private and expandable SRAMs, and a microprocessor controller.

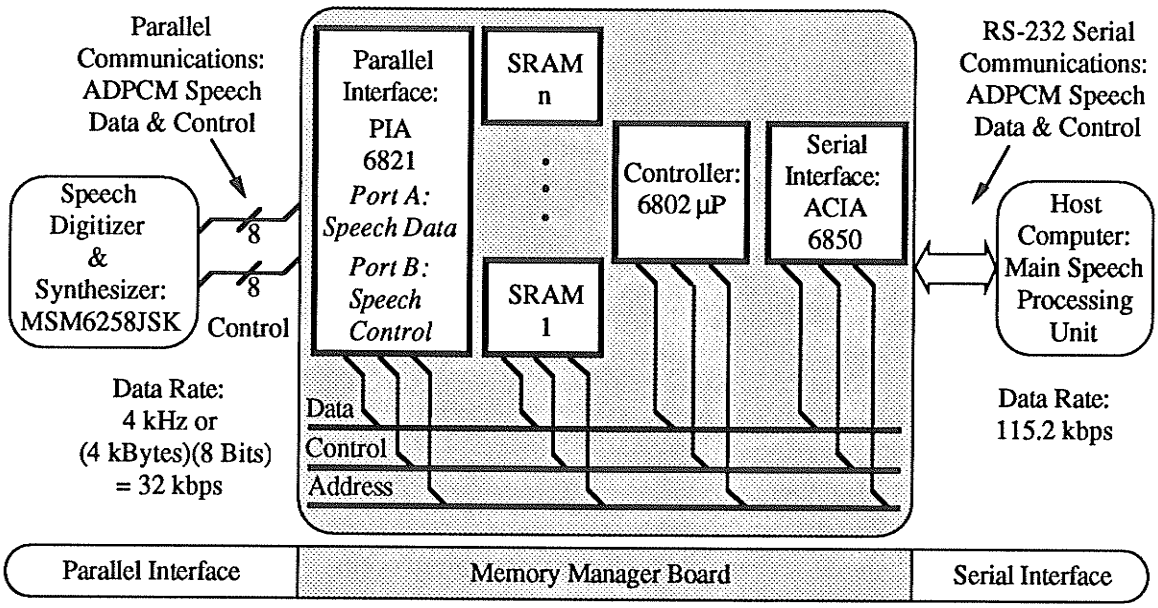


Fig. 4.11 Memory manager block diagram. The memory manager acts as an intermediary device and provides a speech data buffer between the speech processor and the host computer. This provides isolation and allows a continuous and asynchronous flow of data.

4.2.1 Input/Output (I/O) Ports

The memory manager utilizes a dual port configuration for speech data I/O and communication of control signals. The I/O ports are the doorways through which data enters and leaves the buffer device. The physical ports are realized using two interface

adapters: the Peripheral Interface Adapter (PIA) and the Asynchronous Communications Interface Adapter (ACIA). The PIA is used to communicate in parallel with the speech digitizer and synthesizer, while the ACIA is used to communicate serially with the host computer. Both of these devices function as input and output ports. For example, during record mode, speech data is input to the PIA and output to the host via the ACIA. On the other hand, during playback mode, speech data is input to the ACIA and output to the synthesizer via the PIA. Both the PIA and the ACIA are addressable devices, meaning that they are 'turned on' or enabled when their address is selected. The following describes how the PIA and ACIA are used in the memory manager.

4.2.1.1 Parallel I/O: the PIA

This section begins with a brief and general description of the Peripheral Interface Adapter (PIA). A description of the specific implementation of the PIA in the memory manager follows. Finally, the PIA testing procedure and results are given.

4.2.1.1.1 General Description

The Peripheral Interface Adapter (PIA) is a device used for interfacing parallel oriented peripheral devices to the 6800 family of microprocessors (μPs) [Moto83]. The PIA interfaces peripheral devices to the 6800 μP by providing two bidirectional 8-bit data buses for connecting the devices and one 8-bit bidirectional data bus for connecting the μP. Four control lines are provided, two of which are outputs and may be used to control peripheral devices. However, all four control lines are inputs and may be used by peripherals in order to interrupt the μP. In this way, the PIA adapts the electrical characteristics and the number of the signals required by peripheral devices to that required by the 6800 family of μPs.

Internally, the PIA consists of two identical sets of three registers, six registers in total. The A side and the B side both contain one 8-bit data holding register, called Port A and Port B; one 8-bit data direction register, called Data Direction Register of port A (DDRA) and DDRB; and one control register, called Control Register of port A (CRA) and CRB, respectively. Port A and Port B data registers are the physical links between the two data buses connecting the peripheral devices and the data bus connecting the μP.

Each of the six registers are individually addressable. However, only four distinct addresses are required, two for the control registers, CRA and CRB, and two for the data direction registers and the data holding registers, DDRA and Port A and DDRB and Port B. This is because the direction register shares the same address with the holding register. The programming of the control register distinguishes the direction register from the holding register.

The functionality of the PIA is programmable. Programming is achieved by writing coded information into the data direction registers and the control registers. The logic levels of the bits in the data direction register and the control register have specific and functional meaning, as shown in Fig. 4.12 (A more detailed description can be found in [Moto83]). Typically, the direction of the data holding registers are configured first. Then, the functions of the control lines are specified, if required.

Port A and Port B data registers can be configured for either input or output by programming the corresponding data direction register, i.e., DDRA or DDRB. The data direction register is accessed by programming a logic 1 in bit-2 of the control register, as shown in Fig. 4.12. Each bit in the data direction register has a one-to-one correspondence with the associated bit of the data holding register. Writing a logic 1 to any bit of the data direction register configures the corresponding bit in the data holding register for output, whereas, writing a logic 0 configures the bit for input. In general, the data holding register

can be configured for input, output, or not used.

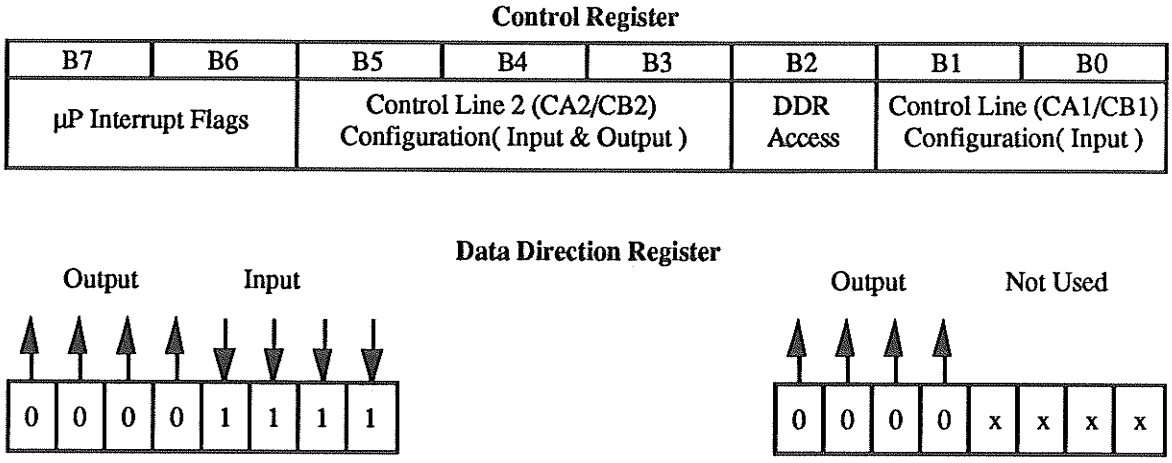


Fig. 4.12 PIA control register (top) and data direction registers (bottom).

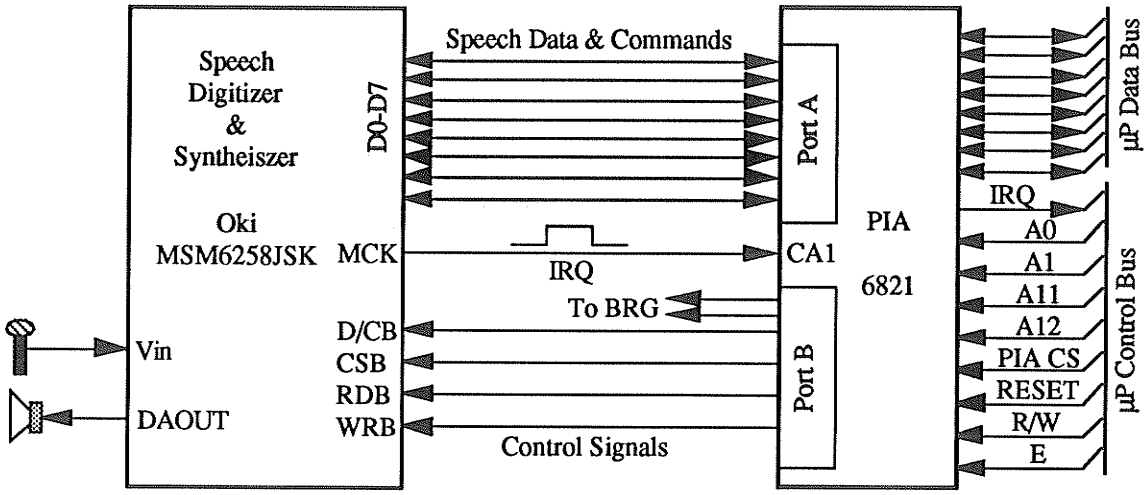
After programming the data direction register, the function of the control lines may be specified by programming the control register. There are two control lines for each side of the PIA, CA1 and CA2 and CB1 and CB2. The output control lines are normally used for controlling a peripheral device, while the input control lines are used for interrupt operation. Typically, for the interrupt type of operation, a peripheral device requests service of the μP by interrupting the μP via the PIA. The peripheral device asserts its interrupt request line, which is connected to one of the input control lines of the PIA. If the PIA is programmed for interrupt operation, that control line of the PIA is connected to the Interrupt ReQuest (IRQ) pin of the μP, and, thus, the interrupt signal is passed to the μP [Bacon86]. Details of exactly how to program the PIA for interrupt operation are given in the discussion of how the PIA is implemented in the memory manager, which is discussed next.

4.2.1.1.2 Implementation

Having briefly discussed the PIA in general, let me now turn to discussing how the PIA is used specifically in the memory manager. The μP controls the speech processor through the PIA. Control signals and commands are issued to the speech processor

through the PIA. Finally, the speech processor interrupts the μP through the PIA.

The schematic diagram of the PIA implementation in the memory manager is shown in Fig. 4.13. The top part of this figure shows the connectivity of the PIA between the



PIA Internal Addressing:

Physical Address	μP Address Line		PIA Control Register Bit		PIA Register Selected
	A0	A1	CRA Bit 2	CRB Bit 2	
1E00	0	0	1	X	Port A
1E00	0	0	0	X	DDRA
1E01	0	1	X	X	CRA
1E02	1	0	X	1	Port B
1E02	1	0	X	0	DDRB
1E03	1	1	X	X	CRB

Fig. 4.13 PIA Implementation and internal addressing.

speech processor and the 6802 μP. The right hand side of the PIA shows the signals that are connected to the 6802 μP, while the left hand side shows the signals that are connected to the peripheral devices, in particular, the speech processor and the Bit Rate Generator (BRG).

There are two groups of signals connecting the PIA to the μP, the control bus and the data bus. The control bus is used to enable, disable, or reset the PIA; to interrupt the μP; to select an internal register; and to perform read or write operations. The bottom portion of Fig. 4.12 shows a table indicating the signals required to address and select one of the six registers within the PIA. The numbers, 1E00, 1E01, 1E02, and 1E03 are the hex representations of the logic levels of the 16-bit μP address bus. The address lines A0 and A1 are the Least Significant Bits (LSBs) of the least significant hex digit of the address lines. In order to address any register, the logic levels of A0, A1, CRA, and CRB must be as that shown in the table. For example, in order to access the DDRA register, the μP must first write a logic 0 to bit-2 of CRA. This distinguishes the direction register from the holding register. Having done this, the μP may then access DDRA by placing the address 1E00 on the address bus. (Note: address lines A11 and A12 are included in the PIA enable logic for added protection).

The data bus is used to read speech data or speech processor status and to write speech data or speech processor commands. The data path connecting the speech processor and the μP is discussed next.

There are three groups of signals connecting the speech processor to the PIA, the control bus, the speech data and command bus, and the interrupt request line. The control bus is connected to Port B of the PIA and is used by two devices, the Bit Rate Generator (BRG) and the speech processor. Bit-6 and bit-7 of Port B are used to select the bit rate of the BRG, whose output is being used by the ACIA (refer to Section 4.2.1.2.2). Bit-0 through bit-3 of Port B are used to control the speech data bus, as described in Section 4.1.2.1. Thus, whenever the μP requires to control the speech data bus, the control code is written to the appropriate bits of Port B.

The speech data and command bus is connected to Port A of the PIA. Thus, whenever the μP requires to read speech data or speech status or to write speech data or speech commands, the data is accessed through Port A.

In Section 4.1.1 it is mentioned that the speech chip provides a *data ready* or a *ready for data* signal, namely, MCK. This signal is intended for use by the μP in order to synchronize its speech data I/O with the speech processor. To this end, MCK is connected to the input control pin, CA1, of the PIA. CA1 is configured in the programming of CRA as an input interrupt signal, and, therefore, CA1 is connected to IRQ of the μP. Thus, each time the speech processor forms an ADPCM byte during recording or each time it is ready to process the next ADPCM byte during playing, it informs the μP of the event by interrupting the μP. Subsequently, the μP responds by either reading or writing speech data from or to the speech processor, respectively.

The programming of the CRA for enabling interrupts via CA1 during recording is different than that during playing. The reason for this is because MCK is signalled differently for record than it is for playback (refer to Section 4.1.1.2). Fig. 4.14 explains the programming of the CRA during record and playback.

CRA Programming During Record

B7	B6	B5	B4	B3	B2	B1	B0
μP Interrupt Flags		x	x	x	1	0	1

B0 = 1 means enable the relaying of interrupt signals to the μP at the occurance of an active edge on CA1.
B1 = 0 means the interrupt flag B7 is set each time CA1 undergoes a high to low transition. A high to low transition of CA1 corresponds to a *data ready* signal issued by the speech processor during record.
B2 = 1 means the access of Port A, data holding register, is enabled.
B3 - B5 = x means don't care.

CRA Programming During Playback

B7	B6	B5	B4	B3	B2	B1	B0
μP Interrupt Flags		x	x	x	1	1	1

B0 = 1 means enable the relaying of interrupt signals to the μP at the occurance of an active edge on CA1.
B1 = 0 means the interrupt flag B7 is set each time CA1 undergoes a low to high transition. A low to high transition on CA1 corresponds to a *ready for data*signal issued by the speech processor during playback.

Fig. 4.14 CRA programming during record and playback.

4.2.1.1.3 Testing

The general testing of the PIA involves writing a 6802 program that alternately writes logic 0 then logic 1 to both data holding registers. The port pins of the PIA are monitored on an oscilloscope. Fig. 4.15 shows the source code of the 6802 PIA test program and the oscilloscope view of Port A. The oscilloscope shows one period of a 5V square wave with period 18 μsec, as expected.

The PIA is also tested in the interrupt mode of operation. The Bit Rate Generator is used to signal a periodic interrupt through the CA1 control line of the PIA. The interrupt routine is similar to the general test program shown in Fig. 4.15, except for two modifications: BRA is replaced by RTI, and RTI is preceded by dummy reads of the port registers (which is necessary to clear the interrupt status bit in the control register).

6802 Pia Test Program

Machine Cycles	Source Code		
2	Loop	LDAA	#$00
5		STAA	Port A
2		LDAA	#$FF
5		STAA	Port A
4		BRA	Loop
total MC=18 x 1 MHz = 18 μsec			

Oscillocope View of Port A and Port B

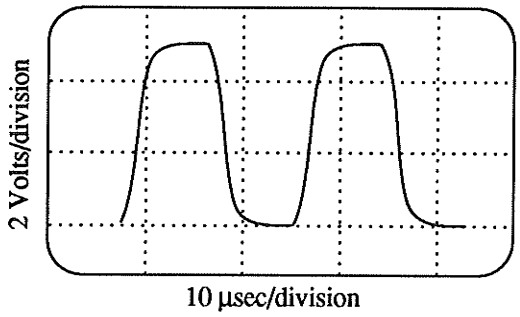


Fig. 4.15 PIA testing.

4.2.1.2 Serial I/O: the ACIA

This section begins with a brief and general discussion of the Asynchronous Communications Interface Adapter (ACIA). A description of the specific implementation of the ACIA in the memory manager follows. Finally, the ACIA testing procedure and results

are given.

4.2.1.2.1 General Description

The ACIA is an offspring of the parent, Universal Asynchronous Receiver and Transmitter (UART), pronounced 'you art'. Motorola's version of the UART is the MC6850 ACIA. The basic purpose of an UART or an ACIA is to interface serial asynchronous data communications to parallel bus organized systems, such as the 6800 family of μPs. These serial asynchronous communications typically originate from another microcomputer system comprising its own UART and μP, as shown in Fig. 4.16.

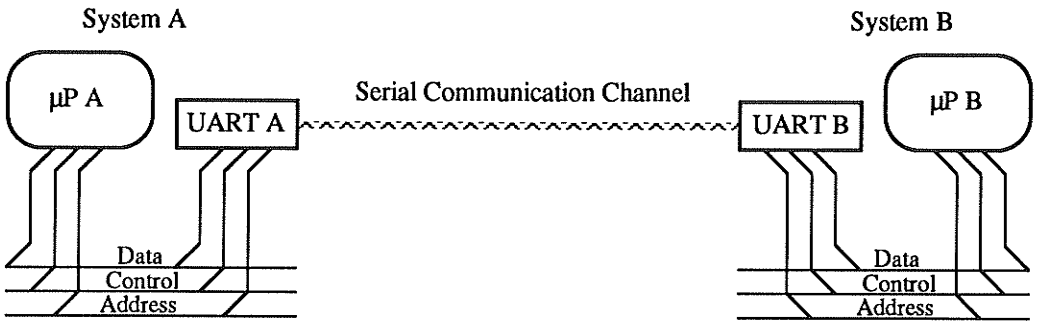


Fig. 4.16 Serial communications system.

The ACIA has two sides, a serial interface and a parallel interface. On the serial side, the ACIA provides for simultaneous bidirectional serial data transfer, called full duplex. This is opposed to half duplex, in which only non-simultaneous directional transfers are possible, such as in CB radio communications. While the end of transmission signalling is possible with full duplex data lines, the ACIA nevertheless includes three control lines for hardware handshaking. On the parallel side, the ACIA includes read/write, enable, and interrupt control lines; register select and chip select address lines; and an 8-bit bidirectional data bus connecting the parallel organized μP. In this way, the ACIA adapts serial formatted data to and from parallel data.

Internally, the ACIA consists of a transmitter, a receiver, a data bus buffer, and a control and status unit. The transmitter section is double buffered, and it consists of a transmit data register, called TxDR, and a transmit parallel-to-serial shift register. Data written to the TxDR register is transferred to the shift register, where it is serialized and, thus, transmitted. This double buffering scheme allows the μP to write the next parallel data to the TxDR register, even though the previous byte may not yet have been totally transmitted.

The receiver section is also double buffered, and it consists of a receive data register, called RxDR, and a receive serial-to-parallel shift register. The double buffering scheme in the receiver section allows the μP to read the RxDR, as the next data byte is being received in the shift register.

The data bus buffer provides the physical link between the μP data bus and the ACIA registers, TxDR and RxDR.

The control and status unit consists of a control register, called CR, and a status register, called SR. The control register is used to program the functionality of the ACIA. The status register is used to obtain status information on a peripheral device, the transmitting and receiving sections, and some error detection control.

The above registers, TxDR, RxDR, CR, SR are individually addressable. However only two distinct addresses are required, one for the transmit and receive registers and one for the control and status registers. What distinguishes the registers is that the TxDR and the CR are write only registers, while the RxDR and the SR are read only registers. In other words, a μP read of the address associated with TxDR and RxDR reads the contents of the RxDR, since RxDR is read only and TxDR is write only.

The functionality of the ACIA is programmable. Programming is achieved by writing coded information into the control register. As shown in Fig. 4.17, the control register controls the transmitter, receiver, interrupt enable logic, and hardware handshaking signals.

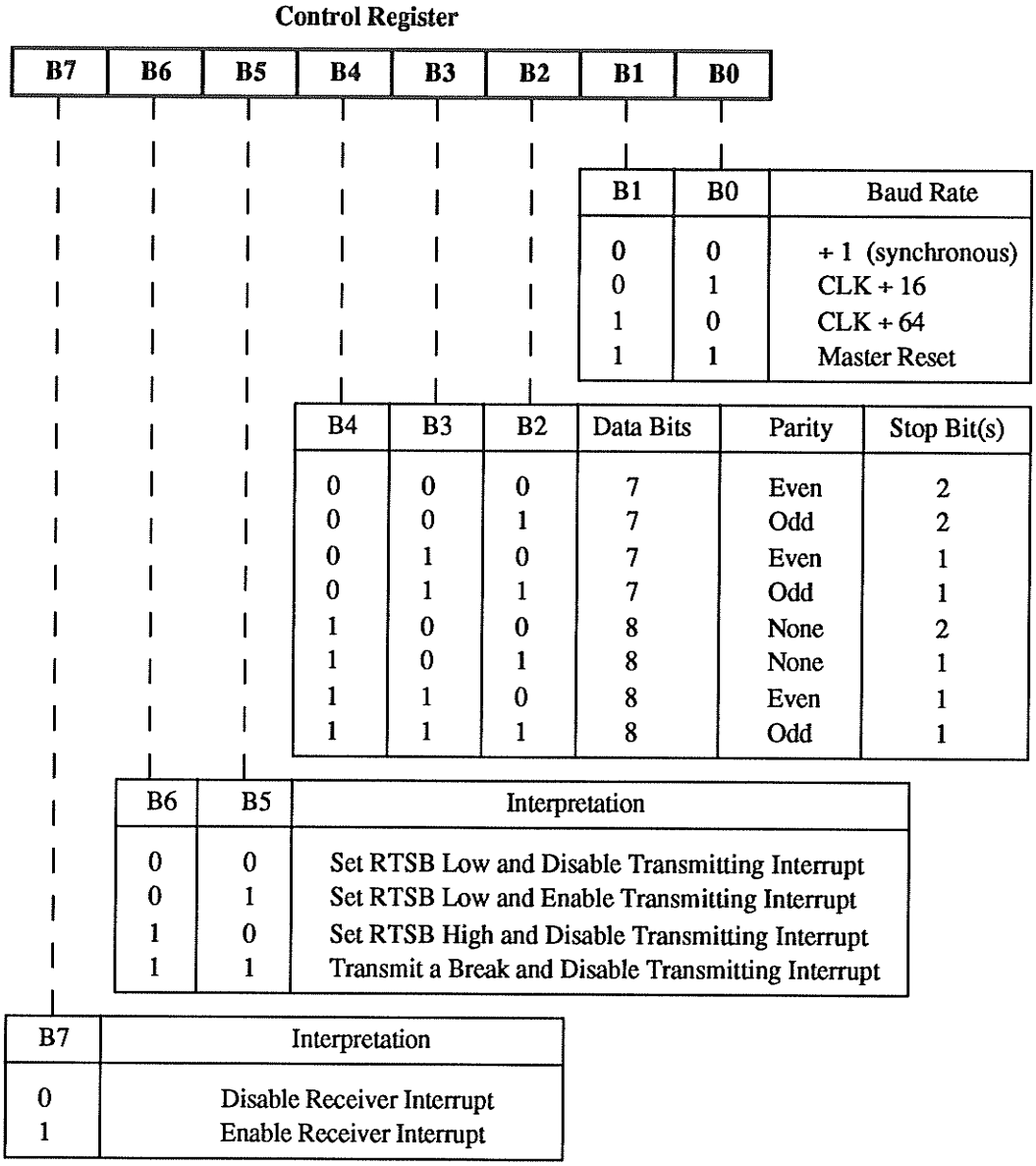


Fig. 4.17 ACIA programmable control register (after [Kins88]).

Bits B0 and B1 control the clocking of the transmitted and received data and, also, reset the ACIA. The ACIA is master reset if both bits are logic 1. The other three logic combinations of these bits are used to generate the clock with which serial data is

transmitted or received. If these bits are both logic 0, the baud rate for the transmitter and the receiver is equal to the clock input at pins TxCLK and RxCLK, respectively. Implicit in this logic is that the other system transmitting or receiving the serial data uses the same clock being input at TxCLK and RxCLK, respectively. Thus, both systems know the start and end of each bit cell being transmitted or received. In this case, the two systems are said to be synchronized. In Figure 4.15, the communication channel would include an additional wire in order to transmit the clock. Because this requires the transmission of the clock between systems, an alternative method for detecting the bit cell is provided. If bits B0 and B1 are logic 1 and 0 or logic 0 and 1, the baud rate of the transmitter and the receiver is equal to the clock input at pins TxCLK and RxCLK divided by 16 and 64, respectively. The receiver section uses the positive transition of RxCLK in order to detect the start and end of each bit cell. What this means is that the receiver takes (RxClk ÷ (16 or 64))/2 samples of the incoming serial data and determines by majority logic the start and end of each bit cell.

Bits B2, B3, and B4 determine the number of bits and the format of each transmitted character. The bit length of each character is the sum of the number of data bits, parity bit (if selected), and stop bit(s). Bits B5 and B6 control the transmitter interrupt logic and, also, define the state of one of the handshaking signals, Request To Send (RTS). Finally, B7 controls the receiver interrupt logic.

The status of the ACIA is available to the μP by reading the status register. This register contains information indicating the status of the peripheral system, transmitter and receiver, and error detection circuits. In particular, the Data Carrier Detect (DCD) and Clear To Send (CTS) inputs indicate whether the receiver is detecting a carrier and indicating a ready to receive (i.e., clear to send), respectively. Also available to the μP are flags, TDRE and RDRF, indicating whether the Transmit Data Register is Empty or whether the Receive

Data Register is Full. Finally, having received data, the μP may check the status of the ACIA internal error detection circuits, which include framing error, overrun error, and parity error.

4.2.1.2.2 Implementation

Having briefly discussed the ACIA in general, let me now turn to describing how the ACIA is used specifically in the memory manager. The purpose of the ACIA in the memory manager is to provide the interface for communicating serial data between the host computer and the 6802 μP. Fig. 4.16 is repeated in Fig. 4.18 to show the specific part played by the ACIA in the overall system. The ACIA receives serial commands from the host computer, such as start record, start playback, and stop. These commands are parallelized and made available to the 6802 μP. The ACIA also serializes and parallelizes speech data to be communicated between the host computer and 6802 μP and, eventually, the speech processor. Finally, the ACIA provides handshaking wires, RTS and CTS, in order that the serial communication of data between the host computer and the memory manager proceeds in a controlled manner.

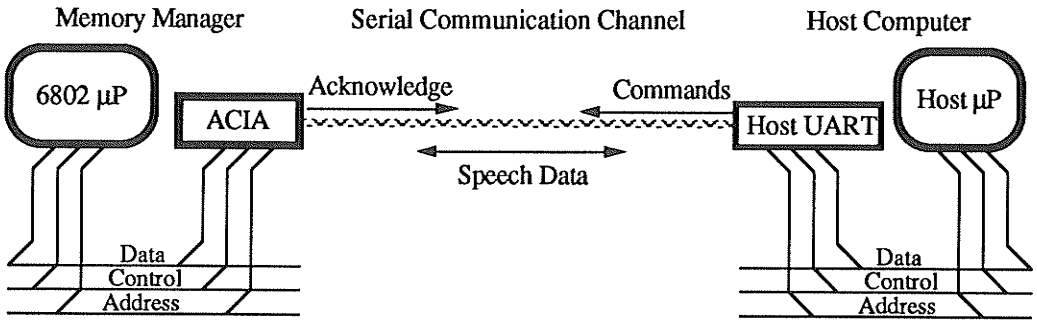
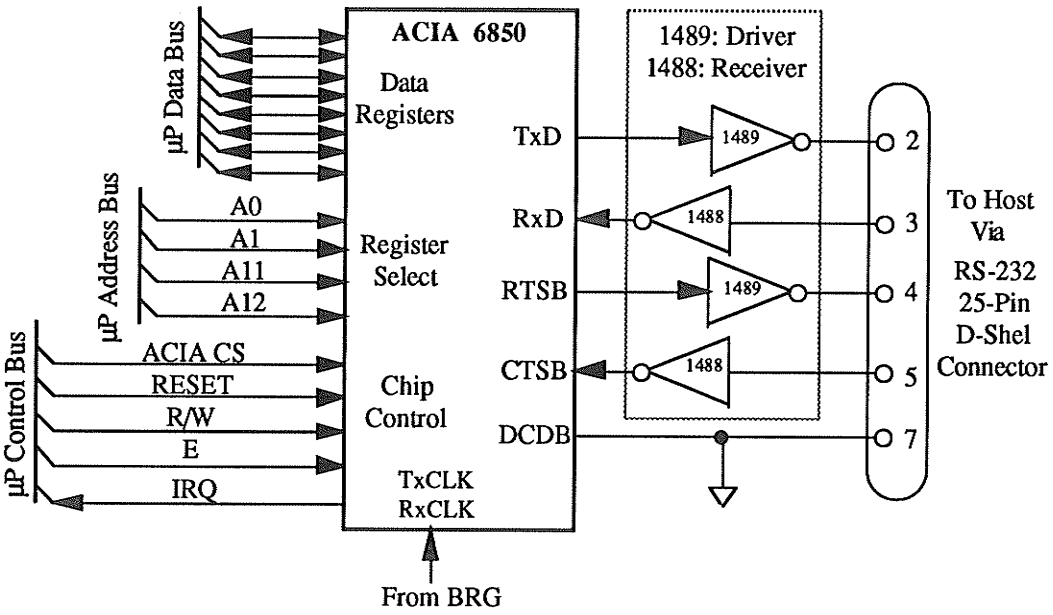


Fig. 4.18 ACIA implementation in serial communication system.

Focussing on the ACIA, Fig. 4.19 shows the schematic diagram and the internal addressing of the ACIA implementation in the memory manager. The top part of this figure shows the connectivity of the ACIA between the 6802 μP and the serial channel connecting

the host computer. There are three sets of signals connecting the μP to the ACIA, the address bus, the control bus, and the data bus. The control bus is used to enable, reset, and select the ACIA. The R/W signal is used in addition to the address bus in order to select internal registers of the ACIA. The bottom part of Fig. 4.18 shows the logic levels required to select one of the four registers, SR, CR, TxDR, and RxDR. Note that the addresses of SR (1D00) and RxDR (1D01) are distinguished from the addresses of CR and TxDR by whether the μP is doing a read or a write, R/W, operation. Finally, the data bus is used by the μP to access data within the ACIA internal registers.



ACIA Internal Addressing:

Physical Address	A0	R/W	ACIA Register Selected
1D00	0	0	Status Register (SR)
1D00	0	1	Control Register (CR)
1D01	1	0	Transmit Data Regiser (TxDR)
1D01	1	1	Receive Data Regiser (RxDR)

Fig 4.19 ACIA schematic diagram (top) and internal addressing (bottom).

There are five signals connecting the host computer to the ACIA, via the serial communications channel. Before entering the channel, the signals TxD, RxD, RTSB, and

CTSB are passed through a driver (1489) and a receiver (1488) [NaSe89]. The purpose of the driver and receiver is to provide the electrical voltage transformation as required by the interface between the ACIA and communications channel, as discussed in Section 4.3.1.

The format of the data transmitted at pin TxD and received at RxD is specified as follows: 115.2, N, 8, 1, IRQ disabled. This means data is to be transmitted at 115.2 kbps; that a character is of length 10 bits and consists of one start bit, eight data bits, no parity bit, and one stop bit. This information is programmed by writing 0101 0101 into the control register. The baud rate is determined by dividing the rate supplied by the BRG by 16, i.e., 16(115.2) ÷ 16 = 115.2 kbps. The transmitter and receiver interrupt capabilities are not required in this application. Therefore, they are disabled.

4.2.1.2.3 Testing

There are different ways to test the ACIA. Perhaps, the simplest method is known as loop back. The idea is to transmit data to the receiver section of the same ACIA. This requires connecting TxD with RxD and RTSB with CTSB. Referring to Fig. 4.19, this can be done by connecting pins 2 and 3 in a loop and pins 4 and 5 in a loop. Testing proceeds with alternatively writing and reading the ACIA and testing whether the received data is exactly the same as that transmitted.

Note that the transmission of serial data can be viewed on an oscilloscope by viewing pin TxD. If the same character is repeatedly sent, then the scope indicates a pseudo square waveform indicative of the group of bit cells being transmitted. If different characters are being sent, flashes of light are viewed on the scope, because the scope cannot trigger onto a non-periodic waveform.

If the loop back tests prove positive, then the next test is to set up a system as that shown in Fig. 4.18. Given that the two systems are identical with respect to the programming of the UART, and, if a similar read test fails, the problem is isolated in the serial communication channel. Perhaps, the wires are not connected properly. Note that TxD from system A must be connected to RxD of system B. Likewise, connect RTSB of system A to CTSB of system B.

The transmission rate of the ACIA can be calculated. The idea is to transmit any arbitrary data. After each character is sent, the μP inverts one of the output lines of a PIA port, say bit 0 of Port A. Viewing the oscilloscope should show a 50% duty cycle periodic square wave of frequency ten times slower than the transmission rate of the ACIA.

4.2.2 Dual Pointer FIFO Buffer

As discussed in Chapter III Section 3.3.2, one of the reasons motivating the memory manager is to allow the host computer the time to process system tasks in the foreground, while communicating speech data with the speech processor in the background. Because the speech processor is a synchronous device, always requiring data or providing data at a constant and periodic rate (i.e., 4 kHz or 32 kbps), some sort of temporary storage or buffer for speech data is required in the event that the host computer is busy doing system tasks when the speech processor provides or requires data. A buffer sufficient for this cause is the dual-pointer First In First Out (FIFO) buffer implemented in software. This section describes both the concept of the dual pointer FIFO and its software implementation in the memory manager.

4.2.2.1 Concept

The dual pointer FIFO buffer implemented in software is a simple concept. It can

be imagined as a circular area in memory, as shown in Fig. 4.20. Initially, the write

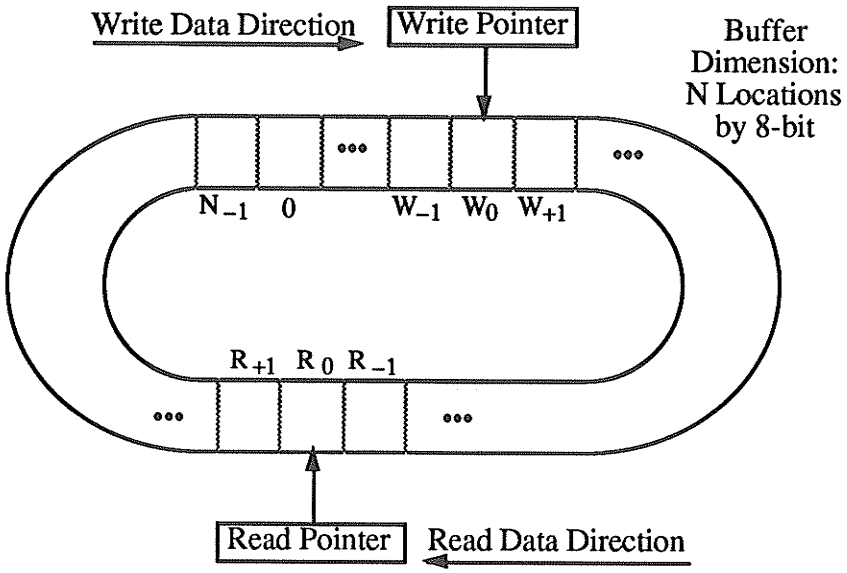


Fig. 4.20 Dual pointer FIFO architecture (after [Kins88]).

pointer points to the same location as the read pointer. Each time data is written into or read from the buffer, the respective software pointer is incremented. Note that this is unlike the hardware FIFO, where, once a data word has been written, that word bubbles down to the next available location [Mono85]. Unread data are never overwritten, so long as the write pointer is always behind the read pointer, or equivalently, the write pointer does not become equal to the read pointer. This implies that writing to and reading from the FIFO are independent and asynchronous events. Furthermore, the rate of reading data from the buffer may be different from that of writing. For instance, given that the rate of writing (f_{write}) is four times as fast as reading (f_{read}) and given the size of buffer, N, then the writing device's waiting time (T_{wait}) between writes is given as follows:

$$T_{wait} = \frac{N}{f_{write} - f_{read}} = \frac{N}{4f_{read} - f_{read}} = \frac{N}{3f_{read}} \tag{4.1}$$

4.2.2.2 Implementation and Testing

In the memory manager the FIFO buffer is implemented using expandable Static Random Access Memories (SRAMs). Currently, two 8K by 8-bit CDM6264 SRAMs are used [RCA84]. These SRAMs physically form the circular type of FIFO buffer, as referred to above. Pointers to the data within the FIFO are implemented using software. Each time data is written to or read from the FIFO, the respective pointer is incremented. When data fills one of the SRAMs, say SRAM A, the write pointer is incremented and then points to the starting location of the other SRAM, SRAM B (note that the SRAMs are placed in contiguous memory locations). When SRAM B becomes full, the write pointer is reset to point at the starting location of SRAM A, and so on. The read pointer is treated similarly.

For the testing procedure given below, the variables in Eq. 4.1 are defined as follows. The time (T_{wait}) corresponds to the time the host computer requires in order to process the received block of speech data and to perform foreground system tasks. The frequency (f_{write}) corresponds to the rate at which the speech processor fills the buffer during recording (e.g., block formation), while (f_{read}) corresponds to the rate at which the μP reads the buffer and sends the speech data to the host computer (e.g., block transmission).

Experiments were conducted in order to determine the size of buffer required to allot the host computer T_{wait}. The objective of the experiment was to test whether the host computer was able to do its background and foreground tasks (the background task is reading speech data and saving speech data to disk, while the foreground task includes refreshing DRAM) before the write pointer became equal to the read pointer. An oscilloscope was set up in order to view the time taken by the host to read speech data and save same to hard disk. The speech processor frequency was f_{read} = 4 kHz = 32 kbps,

while the host computer frequency was 115.2 kbps $\approx 3.6f_{read}$. A software check was set up to assert a hardwired flag if the two pointers became equal. Tests concluded that the size of buffer N = 16K or two 8K SRAMs is sufficient for allotting the host computer T_{wait}.

4.2.3 Memory Map

Section 4.2 mentions that the memory manager is a memory mapped system and that each device is assigned distinct addresses. The descriptions of the PIA (in Section 4.1.1.2) and ACIA (in Section 4.1.2.2) further elaborated on the map by explicitly stating the addresses of each internal register of these two devices. This section describes the memory map of all the devices in the system.

Table 4.1 shows the memory map of the memory manager. The dimension of the available memory space is given by the number of address lines, 16, and data lines, 8, used by the 6802 μP. Therefore, the dimension is 64K by 8-bit. The memory map consists of memory devices and peripheral devices. The memory devices include one 2K by 8-bit Erasable Programmable Read Only Memory (EPROM), six 8K byte Static Random Access Memories (SRAMs), and 256 bytes of internal Random Access Memory (RAM). The peripheral devices include one Peripheral Interface Adapter (PIA) and one Asynchronous Communication Interface Adapter (ACIA). Note that each of these devices have distinct memory locations.

The EPROM is located at the top of the map, and it is used to store the 6802 program and any other system tables, variables, or parameters. The contents of this memory chip are referred to as system firmware. This chip requires 2K bytes of memory, and it is assigned locations $F800_{16}$ through $FFFF_{16}$.

It is interesting to note why the EPROM is located at the top of the memory map. The main reason is that the 6802 uses the locations $FFF8_{16}$ to $FFFF_{16}$ as interrupt vectors [Moto83]. The addresses of the corresponding interrupt service routines must be burnt into

Table 4.6 Memory map of the memory manager.

Memory Address (Hex)	Device	Address bit								
		A15	A14	A13	A12	A11	A10	A9	A8	A7-A0
FFFF F800	2K EPROM	1	1	1	1	1	X	X	X	X
E000	Not Used									
DFFF C000	8K SRAM1	1	1	0	X	X	X	X	X	X
BFFF A000	8K SRAM2	1	0	1	X	X	X	X	X	X
9FFF 8000	8K SRAM3	1	0	0	X	X	X	X	X	X
7FFF 6000	8K SRAM4	0	1	1	X	X	X	X	X	X
5FFF 4000	8K SRAM5	0	1	0	X	X	X	X	X	X
3FFF 2000	8K SRAM6	0	0	1	X	X	X	X	X	X
1F00	NOT USED	0	0	0	1	1	1	1	1	X
1E00	PIA	0	0	0	1	1	1	1	0	X
1D00	ACIA	0	0	0	1	1	1	0	1	X
0080	NOT USED									
007F 0000	INTERNAL RAM	0	0	0	0	0	0	0	0	X

the EPROM by the designer at their respective locations. For example, The memory manager's main program resides in memory starting at $F800_{16}$. This is the program that is run when the memory manager circuit is powered up. Therefore, the designer must load the address $F800_{16}$ in the reset vector location, which is $FFFE_{16}$ to $FFFF_{16}$. Furthermore, if

the system is interrupt driven, as is the memory manager, the designer must load the starting location of the interrupt routine in the interrupt vector location, which is $FFF8_{16}$ $FFF9_{16}$.

Allocated directly below the EPROM is space for six SRAM chips. Each SRAM requires 8K bytes of memory, and they are assigned contiguous addresses, ranging from 2000_{16} to $DFFF_{16}$. In the memory manger, only two 8K SRAMs are connected. However, if more are required, they may be easily installed, since space has been allocated.

The peripheral devices (PIA and ACIA) are also memory mapped devices with base addresses $1E00_{16}$ and $1D00_{16}$, respectively. They are discussed in Section 4.2.1.

A special feature of the 6802 μP is the 128 bytes of internal RAM addressable at locations 0000_{16} to $007F_{16}$. Data access is more efficient because only eight bits are required to address these locations. This memory may be used automatically by the μP as the stack for saving the context of the machine upon an interrupt, or it may be used by the programmer as an internal scratch pad. The memory manager uses this area of memory for stack operations, and, during the initialization section of the main code, the stack pointer is loaded with the starting address of the stack, $007F_{16}$.

4.2.3.1 Memory Decoding

The memory manager contains a memory decoder. The purpose of the memory decoder is to decode the address bus. The memory decoder inputs address lines and outputs chip select signals, which are fed to enabling pins of peripheral devices. All devices described in the above memory map are connected to the same data bus and share this bus with the μP. If more than one of these devices were enabled and attempted to assert different voltage levels on the data lines at any one instant, data on the bus would be invalid. It is important to ensure that only one device is driving the data bus at any one time, in order

to prevent bus contention. The memory decoder acts as an intermediary device between the μP and the devices connected to the μP. When the μP requires access of a certain device, then the address of that device is placed on the address bus. The decoder intercepts the address bus, determines which device the address refers, and enables that device while disabling other devices. If all devices connected to the μP had 16 input chip select lines, then a decoder would not be required, because only one device's address would be on the address bus at a time. However, most devices do not have 16 chip select lines, because it is impractical.

Figure 4.21 shows the schematic diagram of the memory decoder in the memory

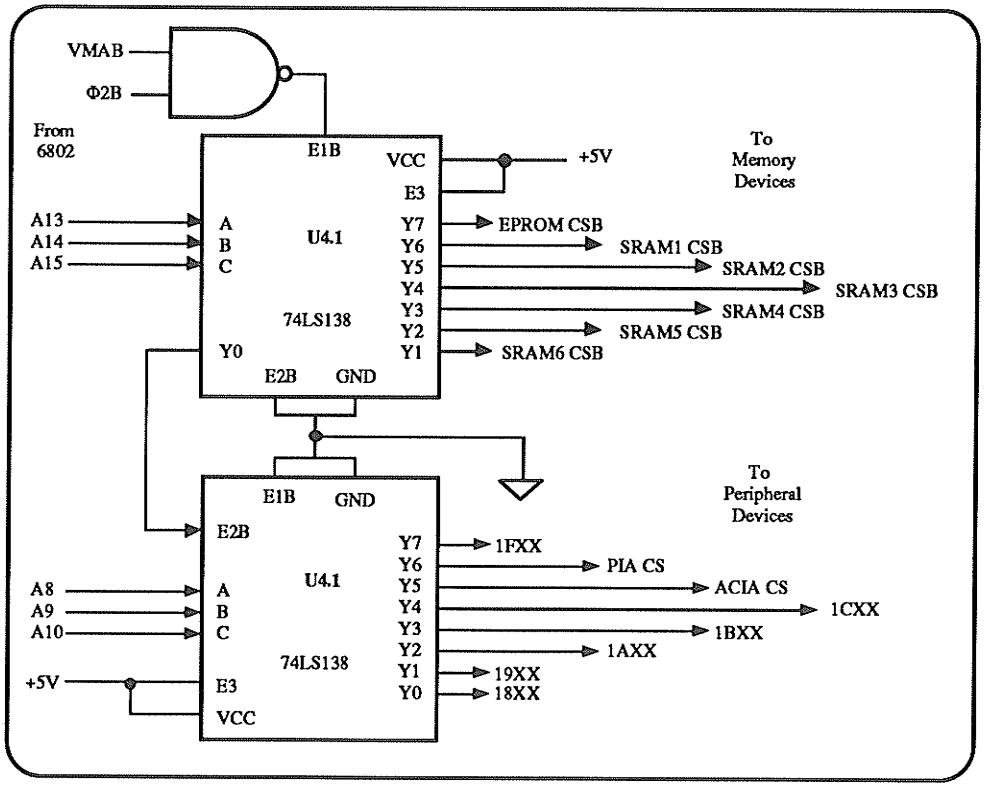


Fig 4.21 Memory manager memory decoder.

manager. The circuit is implemented using two 1-of-8 74LS138 [Texa84] decoders. Note that the circuit inputs the address lines A8, A9, A10, A13, A14, and A15, while outputting

chip select signals for the EPROM, each of the SRAMs, the PIA, and the ACIA. Note also that the enable of both decoders depends on the logical NAND of Valid Memory Address (VMA) and Enable (E). E is phase 2 or Φ2 of the 6802 clock. The designers of the 6800 family of microprocessors intended Φ2 to be used as a data valid signal, i.e., Φ2 is asserted when data on the data bus is valid. However, in order to distinguish between memory referenced and non-memory referenced instructions, the VMA signal is used. During non-memory referenced instructions of the 6800 family of microprocessors, the address lines are unstable and, therefore, may cause erroneous chip selects. Thus, a memory device is being accessed only when VMA and Φ2 are logic 0. This is assumed in the following discussion of EPROM, SRAM, PIA, and ACIA chip selection.

When each of the lines A13, A14, and A15 are logic 1, the EPROM is selected. In fact, all that is required to decode the EPROM is A13, A14, and A15 being logic 1, because, as shown in Table 4.6, no other device is being addressed when A13, A14, and A15 are logic 1.

The allocation of the EPROM at the top of the memory map places a constraint on the allocation of the six SRAMs. Since the SRAMs are 8K in size, 13 address lines (A0 to A12) are required to specify the 8192 distinct memory cells. This leaves three address bits for decoding, namely, A15, A14, and A13. Since the combination A13 = 1, A14 = 1, and A15 = 1 has been used for the EPROM, the first SRAM must be placed at the address leading with A13 = 0, A14 = 1, and A15 = 1. Once again, all that is required to decode the SRAMs is the three address lines A13 , A14 , and A15, because no other device is being addressed when A13, A14, and A15 have the logic values corresponding to the addresses of the six SRAMs.

The PIA and the ACIA are selected when A13, A14, and A15 are logic 0. Note that this is the only remaining combination of A13, A14, and A15. As shown in Fig. 4.21, when

A13, A14, and A15 are logic 0, Y0 is active, and it enables the second decoder. The PIA and ACIA are distinguished in the second decoder by the logic levels on address lines A8, A9, and A10.

The other chip select signals available in the decoder circuit may be used for expansion.

4.2.4 Controller: the 6802 μP

The controller of the memory manager is implemented using the Motorola 6802 μP [Moto83]. The 6802's main purpose is to control communications between its external devices, the host computer, and speech processor. This communication includes the transmission of commands, acknowledgements, and speech data. Because the communication of these data is asynchronous and occurs at different rates, the 6802 acts as a data traffic controller. While allowing the speech processor to communicate its synchronous data at a constant rate, the 6802 allows the host computer to communicate its asynchronous data at a different and much faster rate.

The 6802 controller achieves its purpose by controlling and making effective use of its peripheral devices, the ACIA, PIA, and FIFO buffer. The ACIA is used to communicate serial data with the host computer. The PIA is used to communicate parallel data with the speech processor. And the FIFO buffer is used to implement a simple form of pipelining. This buffer provides temporary storage of speech data when the host computer is busy doing other system and foreground tasks.

4.2.4.1 Hardware Implementation

The 6802 μP comes in a 40 pin Dual In line Package (DIP). In order to test a circuit in which the μP is applied, the chip must be physically wired into the circuit and the code for the software program must be burnt into a Read Only Memory (ROM) chip each time a new program is written.

4.2.4.1.1 6802 Emulator: EM-186

However, for developing and debugging purposes, it is convenient to use an emulator. One such device is the EM-186 6800/6802 diagnostic emulator [ApMi85]. The emulator allows the programmer to develop and debug a target system without having to physically insert the μP into the circuit or to burn code into a ROM chip. All of the internal registers, accumulators, functions, instructions, addressing modes, and special features offered by the μP are emulated by the emulator. Furthermore, the EM-186 has 64K of Random Access Memory (RAM) built in. This enables the programmer to download code for a program into emulator RAM, rather than having to burn the code into a ROM chip. In other words, the built in RAM acts as ROM. Once the debugging phase is completed and the target system verified to be working properly, then the physical insertion of the μP and code burning proceeds naturally.

The EM-186 is simple to use. Some of its features include a serial port to download code, breakpoint and instruction tracing, memory and diagnostic tests, and display of internal registers, accumulators, and flags.

In order to download code, a communications package, such as ProComm, running on a host computer transmits the code to the emulator via the RS-232C compatible serial port. In order to invoke the download mode of the emulator, the codes E1 followed C3 are

typed into the emulator keyboard. The code, E1, is an acronym for Enable serial port 1. C3 is the code that places the EM-186 into the serial input mode. The baud rate is selected via a bank of switches located at the back of the emulator. The program is placed in emulator RAM at the location specified by the ORG directive in the source listing. If errors occur during transmission, the emulator gives (beeps) a warning message.

In order to run and test the program, the starting address of the program must be loaded into the program counter. This is done by using the appropriate emulator commands typed into the emulator keyboard. Furthermore, this can be done by either loading the program counter or by loading the starting address of the program into the reset vector location. The latter method is more practical, since, when the power of the target system is turned on, a global reset occurs and, in particular, the 6802 program counter is automatically loaded with the address contained within the reset vector.

When the target system is powered up, the program may be tested by using either of the emulator commands, run, run until breakpoint, or step. Breakpoint and stepping are two very powerful debugging tools. Each time a breakpoint occurs or each time an instruction is stepped through, the EM-186 transmits the contents of internal μP registers, accumulators, and flags through the serial port to the host computer, where they can be viewed on the screen.

The EM-186 also provides memory and some diagnostic tests. For example, in the FIFO buffer of interest, the SRAMs are tested. In particular, invoking the emulator command A1, along with specifying a starting and ending address, the emulator writes zeroes and ones to the starting location, attempts to read same, then continues with every location up to the ending address. If an error occurs during reading, the emulator beeps and displays the location and problematic data. This is convenient for testing whether the wiring

of devices in the target system is correct.

4.2.4.2 Software Implementation

The code of the memory manager controller is written using the 6802 instruction set. The code is given in Appendix A. The code consists of system initialization, command reception, record, playback, and stop routines. These routines are sequentially executed. After system initialization, the 6802 executes command reception, where it waits for a valid command. Having received a valid command, that command is executed until a stop command is sensed, upon which, the controller executes the stop routine. The stop routine vectors the controller back to initialization, and the whole procedure is repeated.

4.2.4.2.1 Initialization

The initialization routines initialize variables, the PIA, and the ACIA. Variable initialization includes setting the read and write FIFO buffer pointers, Read_Ptr and Write_Ptr, respectively to an arbitrary starting location, 2000_{16}, which happens to be the starting address of the first SRAM.

The PIA is initialized as mentioned in Section 4.2.1.1.2. Port B is configured for output in order to facilitate supplying control signals to the speech processor and BRG. More specifically, 83_{16} = 1000 0011_2 is written to Port B. B7 and B6 are input to the BRG. B3 and B2 reset to logic 0 enables and selects the command input mode of the speech processor, respectively. B1 and B0 sets RD and WR to logic high, respectively. Port A is configured for speech chip command/status and speech data I/O. Whether Port A is input or output depends on the forthcoming command issued by the host computer. The interrupt logic is set to enable active transitions on CA1. Thus, the controller services the speech processor by responding to speech processor generated interrupts. Refer to page

A16 for PIA initialization code and comments.

The ACIA is initialized as mentioned in Section 4.2.1.2.2. It is master reset and the initial state of RTSB is high (inactive). The serial side is set up to communicate data with the following attributes: 115.2, N, 8, 1. The ACIA is not interrupt driven. The controller polls the ACIA during command reception, record, and playback routines. Refer to page A15 for ACIA initialization code and comments.

4.2.4.2.2 Command Reception

Figure 4.22 shows the flow chart for the command reception routine. This routine is executed following system initialization. After the ACIA has been reset and RTSB set to the inactive state (logic high), the controller determines whether the host is requesting to send by looking at B3 of the ACIA status register. If the host is requesting to send, the controller acknowledges the request by giving the host clearance to send, i.e., by resetting RTSB to logic 0 (active). The controller then reads a byte of data from the ACIA receive data register and compares the byte to command codes in order to determine whether the byte of data is a valid command. If the command is valid, the controller executes the appropriate routine. If an error occurs, the controller vectors back to the start of the program, executes the initialization routine, and attempts the command reception routine once again.

4.2.4.2.3 Playback

The playback routine consists of subroutines and groups of code that can be described as command acknowledge, speech chip playback initialization, buffer pre-load, foreground speech data fetch, and background speech data play.

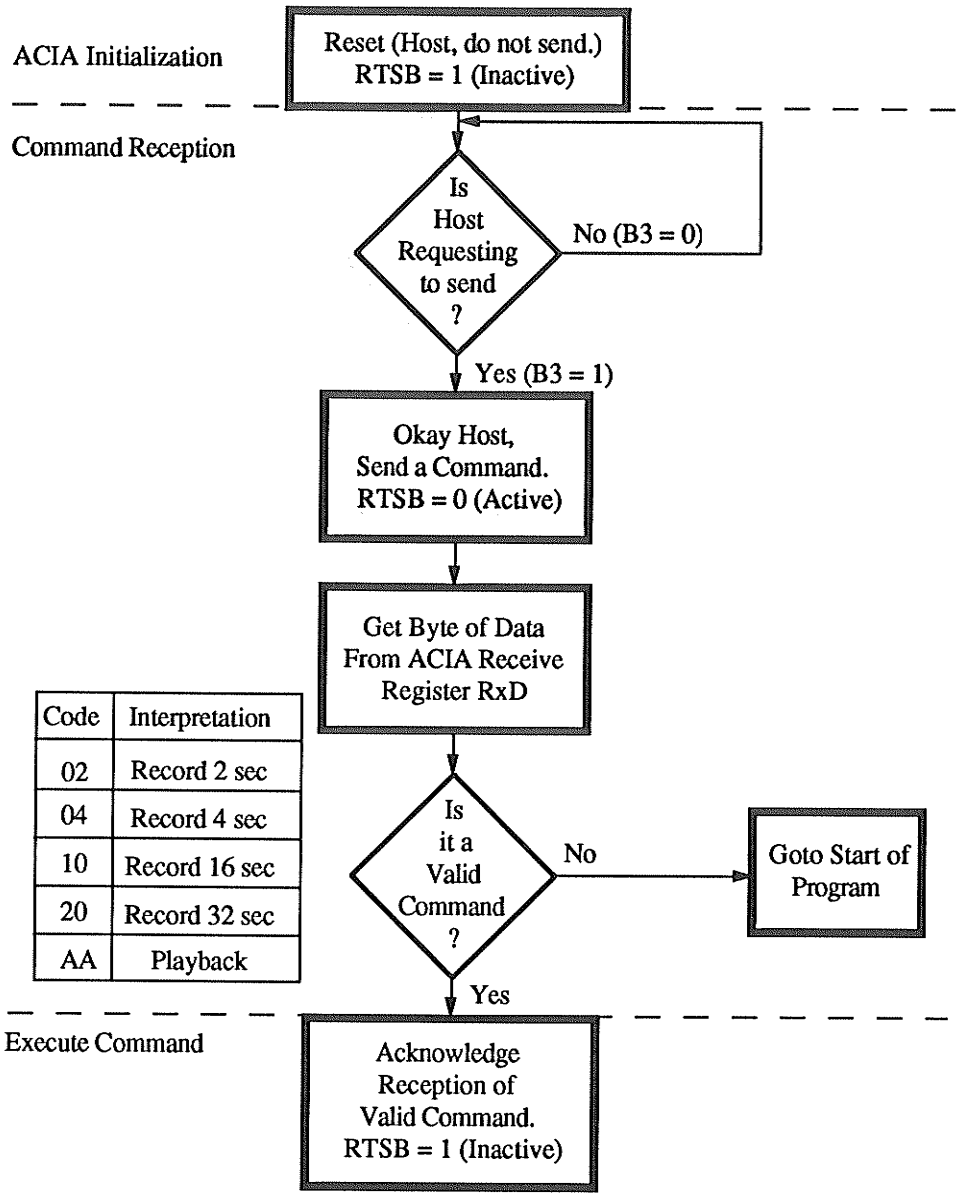


Fig. 4.22 Command reception flow chart.

Having received a valid playback command, the controller jumps to its playback routine, where the first subroutine executed is command acknowledge. In this subroutine, the controller sets RTSB to logic 1 (inactive) in order to inform the host that a valid command has been received. The controller then waits for an acknowledge signal by monitoring B3 of the ACIA status register.

When the host signals an acknowledge, the controller initializes the speech chip for playback. In this subroutine, the controller writes the playback command to the speech chip. Playback command is written to the speech chip by placing the command code on the data bus and then providing the sequence of appropriate logic levels on the input control signals WR, RD, D/CB, and CS. The playback command code, 02_{16}, is written to Port A of the PIA. This places the command code on the speech data bus. Following this, 82_{16} = $1000\ 0010_2$ is written to Port B, of which, bits B0 to B3 are connected to WR, RD, D/CB, and CS, respectively. This places WR to logic 0. About 12 μsec thereafter, $82_{16} = 1000\ 0011_2$ is written to Port B. This places WR to logic 1 and, thereby, writes the playback command to the speech chip.

Also in the playback initialization routine, the PIA is enabled to relay interrupts to the 6802 via CA1 low-to-high transitions. This is done by writing $07_{16} = 0000\ 0111_2$ to CRA of the PIA.

After the speech chip is initialized for playback, the controller executes a buffer pre-load subroutine called FillQueue. The purpose of this subroutine is to fill the start-up queue of the FIFO buffer. This is necessary in the event the speech processor interrupts the controller, asking for a byte of speech data, before the controller has yet received the first byte from the host computer. A size of 256 bytes for the queue is sufficient for starting the process of playback, and the duration of this size is perceptually unnoticed.

As shown in Fig. 4.23, this routine begins with handshaking. The controller determines whether the host is requesting to send. If so, the controller acknowledges the request by placing RTSB to logic 0 (active). Henceforth, the transmission of each byte is controlled by RTSB. Whenever RTSB is logic 0, the host knows it can transmit speech data, and whenever RTSB is logic 1, the host knows it must not transmit. In this routine, the controller resets RTSB to logic 0 when it is ready to receive a byte and sets RTSB to logic 1

after receiving each byte. After receiving each byte, the controller, checks whether the queue is full. If the queue is not full, the controller continues receiving data, whereas, if the queue is full, the controller concludes this routine and begins execution of the next routine, main playback.

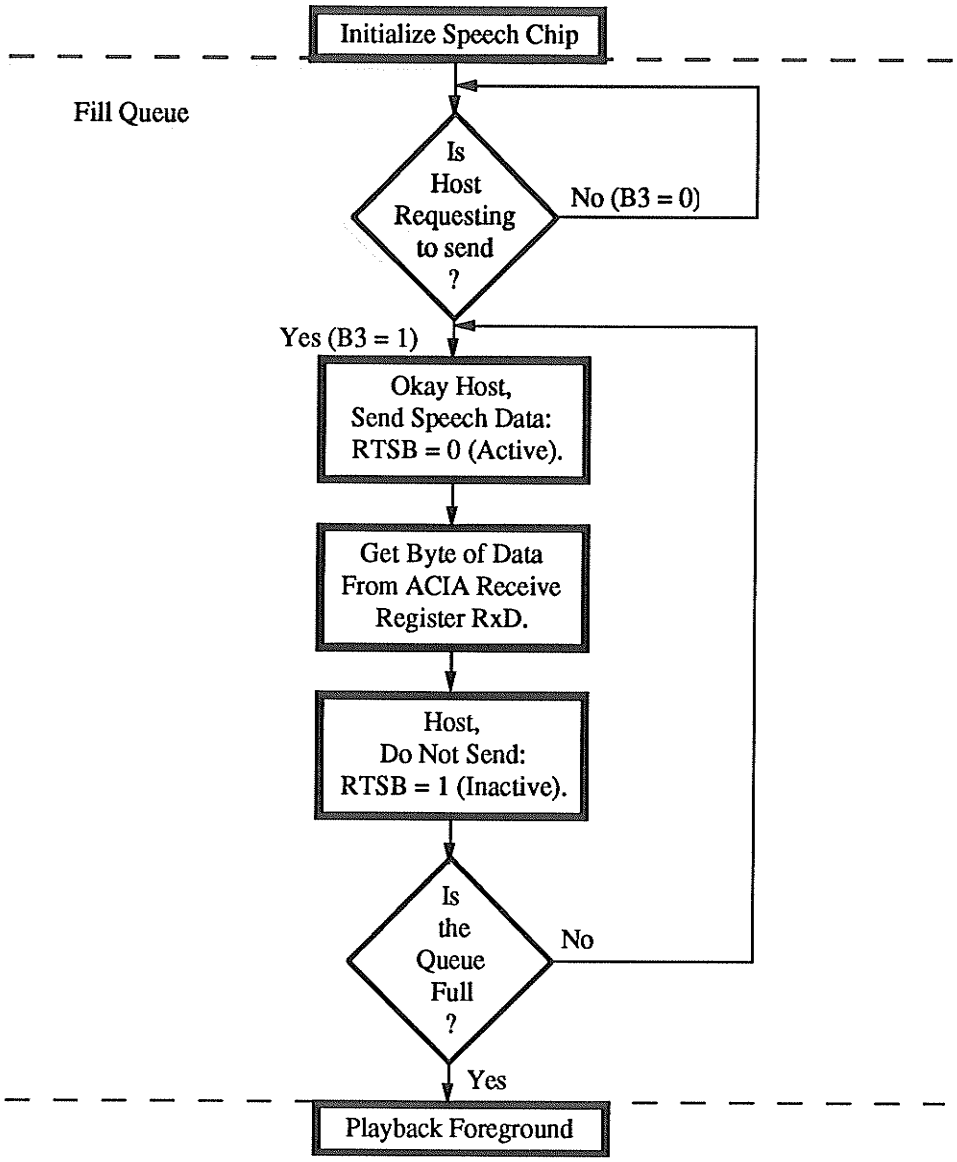


Fig. 4.23 Fill queue flow chart.

The main playback routine consists of foreground and background processing. In the foreground, the controller continually receives data from the host and writes these data to the FIFO buffer. The FIFO write pointer (Write_Ptr) points to the next available location

in the buffer. In the background, the controller reads speech data from the FIFO and writes these data to the speech chip. The FIFO read pointer (Read_Ptr) points to the next byte of speech data in the buffer.

Without interruptions, foreground processing is capable of receiving a byte of speech data and writing it to the buffer approximately every 86 μsec. However, background processing interrupts the foreground every 250 μsec. Therefore, during foreground processing, the controller receives almost two bytes for every one byte requested by the speech chip during background processing. Note that the remaining time (250 – 2[86] = 78 μsec) corresponds to CPU time. As a consequence, in order to detect the buffer full condition, the controller monitors Write_Ptr and Read_Ptr for equality.

As shown on Fig. 4.24, playback foreground processing begins with determining whether the host is requesting to send. If so, Write_Ptr is compared with Read_Ptr. If they are equal, this means the buffer is full, and the host is informed temporarily not to send data. If the buffer is not full, the host is informed to send. Note that the current state of RTSB is saved whenever it is changed. After receiving each byte, the controller increments Write_Ptr. The controller executes the above steps continually until it is interrupted by the speech processor.

Background processing is caused by speech chip initiated interrupts. Each 250 μsec, when the speech chip is ready to convert the next digital ADPCM byte to the analog domain, the speech chip interrupts the 6802 controller via CA1 of the PIA. During each interrupt routine, the controller writes one ADPCM byte to the speech chip. Like foreground processing, the controller checks whether the FIFO read pointer (Read_Ptr) is pointing to the same location as Write_Ptr. Unlike foreground processing, the interpretation of the event when the two pointers point to the same location is that the buffer is empty. Furthermore, the empty condition can only have occurred because the host quit

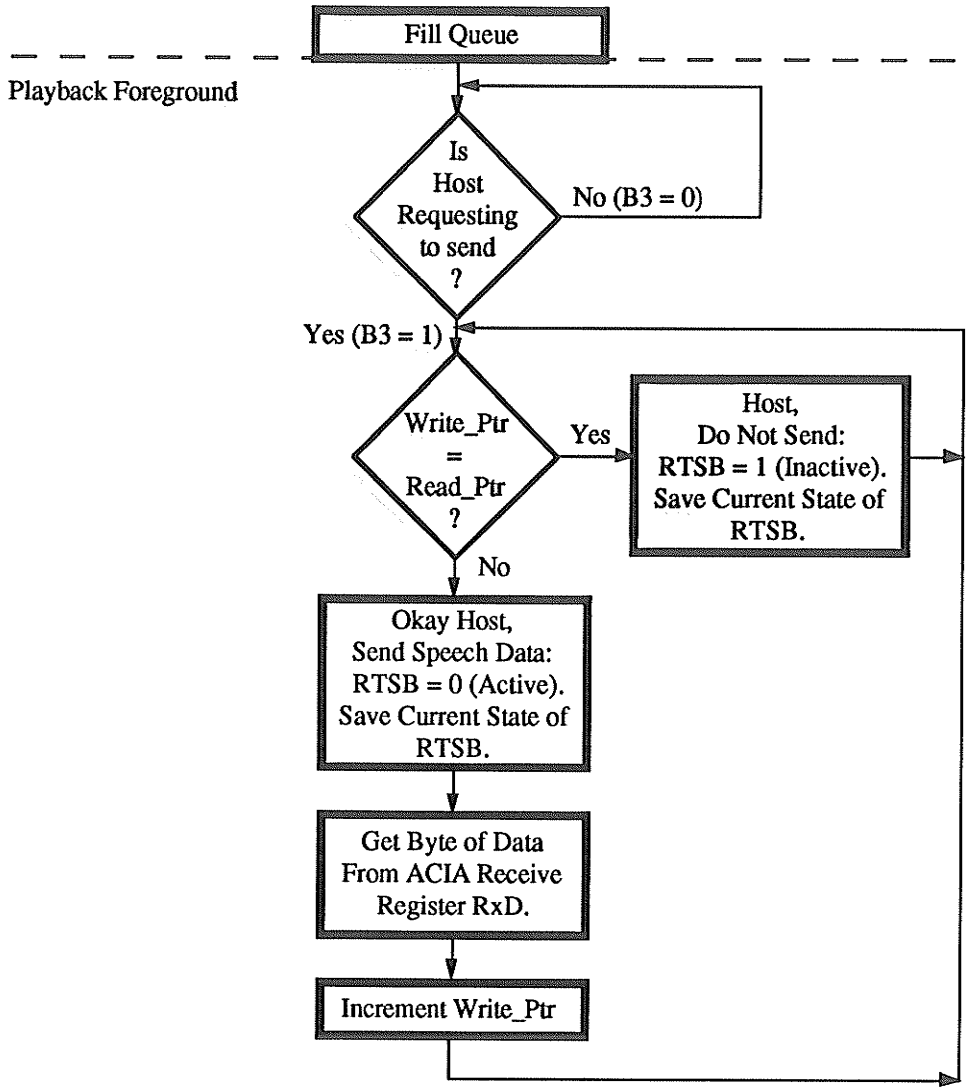


Fig. 4.24 Playback foreground processing flow chart.

sending data. This condition is interpreted as a stop command issued by the host computer.

As shown in Fig. 4.25, background processing begins with informing the host not to send. In other words, the controller cannot temporarily receive data from the host because the speech chip requires service. Recall that the state of RTSB prior to each interrupt is saved in foreground processing. After setting RTSB to logic 1, the controller writes one byte to the speech chip. In more detail, the ADPCM byte pointed to by the FIFO read pointer (Read_Ptr) is written to Port A of the PIA. This places the byte on the speech

data bus. Following this, $86_{16} = 1000\ 0110_2$ is written to Port B, of which, bits B0 to B3 are connected to WR, RD, D/CB, and CS, respectively. This enables the chip, selects the data mode, and places WR to logic 0. About 12 μsec thereafter, $87_{16} = 1000\ 0111_2$ is written to Port B. This places WR to logic 1 and, thereby, writes the ADPCM byte to the speech chip. After data is written to the speech chip, the controller checks whether Read_Ptr is equal to Write_Ptr. If true, the controller interprets this as an implied stop command and jumps to the stop routine. If false, Read_Ptr is incremented to point to the next ADPCM byte in FIFO buffer, the state of RTSB prior to the interrupt routine is restored, and the controller returns to the foreground routine (RTI).

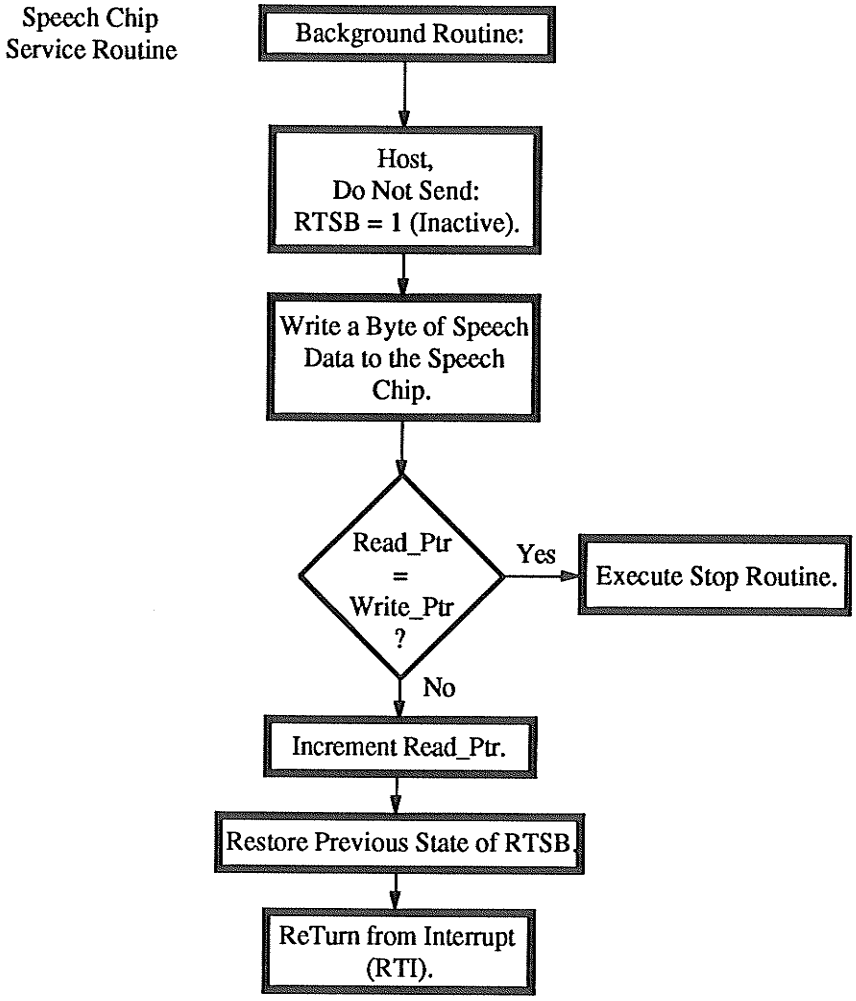


Fig. 4.25 Playback background processing flow chart.

4.2.4.2.4 Record

The record routine consists of subroutines and groups of code that can be described as command acknowledge, speech chip record initialization, foreground speech data write, and background speech data record.

Having received a valid record command, the controller jumps to its record routine, where the first subroutine executed is command acknowledge. In this subroutine, the controller sets RTSB to logic 1 (inactive) in order to inform the host that a valid command has been received. The controller then waits for an acknowledge signal by monitoring B3 of the ACIA status register.

When the host signals an acknowledge, the controller initializes the speech chip for record. In this subroutine, the controller writes the record command to the speech chip. Record command is written to the speech chip by placing the command code on the data bus and then providing the sequence of appropriate logic levels on the input control signals WR, RD, D/CB, and CS. The record command code, 04_{16}, is written to Port A of the PIA. This places the code on the speech data bus. Following this, $82_{16} = 1000\ 0010_2$ is written to Port B, of which, bits B0 to B3 are connected to WR, RD, D/CB, and CS, respectively. This places WR to logic 0. About 12 μsec thereafter, $83_{16} = 1000\ 0011_2$ is written to Port B. This places WR to logic 1 and, thereby, writes the record command to the speech chip.

After initializing the speech chip for record, the controller begins the main record routine. The main record routine consists of foreground and background processing. In the foreground, the controller continually reads data from the FIFO buffer and transmits these data to the host. The FIFO read pointer (Read_Ptr) points to the next byte of speech data in the buffer. In the background, the controller reads speech data from the speech chip and writes these data to the FIFO buffer. The FIFO write pointer (Write_Ptr) points to the

next available location in the buffer.

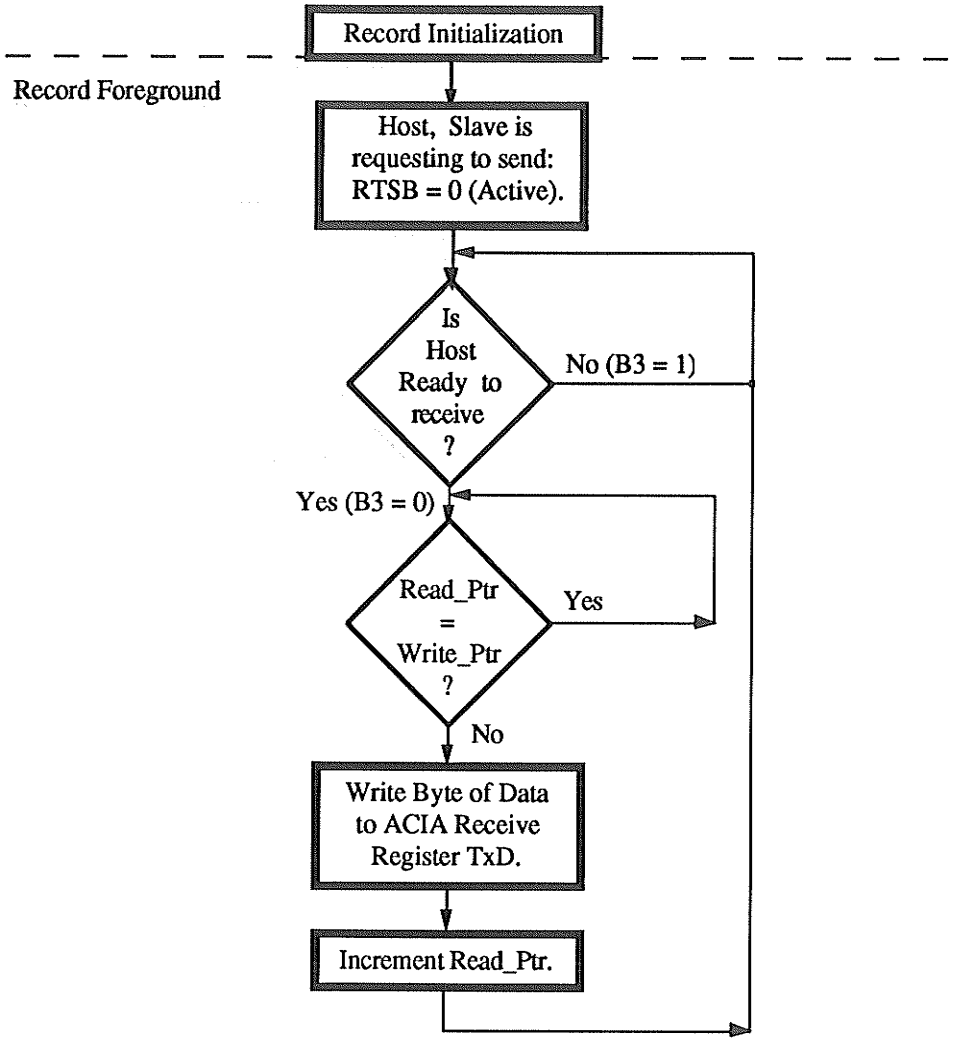


Fig. 4.26 Record foreground processing flow chart.

As shown in Fig. 4.26, the foreground of record processing begins with the controller informing the host of a request to send. After this, the controller waits for an acknowledge, indicating the host is ready to receive data. After the acknowledge is sensed, Read_Ptr is compared with Write_Ptr in order to determine whether the buffer contains any speech data. The condition when Read_Ptr points to the same location as Write_Ptr indicates the buffer is empty. This is so because the host is capable of reading serial speech data about twice as fast as the speech processor produces parallel speech samples. If the condition is false, the controller sends a byte of speech data to the host, increments the read

pointer, and branches back to determining whether the host is ready for the next byte. The controller executes the above steps continually until it is interrupted by the speech processor.

Background processing is caused by speech chip initiated interrupts. Each 250 μsec, when the speech chip indicates that a digital ADPCM byte is ready for reading, the speech chip interrupts the 6802 controller via CA1 of the PIA. During each interrupt routine, the controller reads one ADPCM byte from the speech chip. Like foreground processing, the controller checks whether the FIFO read pointer (Read_Ptr) is pointing to the same location as Write_Ptr. Unlike foreground processing, the interpretation of the event when the two pointers point to the same location is that the buffer is full. Furthermore, this full condition can only have occurred because the host quit recording data. This condition is interpreted as a stop command issued by the host computer.

As shown in Fig. 4.27, the background of record processing immediately begins with reading one byte of speech data from the speech chip. In more detail, 85_{16} = 1000 0101_2 is written to Port B, of which, bits B0 to B3 are connected to WR, RD, D/CB, and CS, respectively. This enables the chip, selects the data mode, and places RD to logic 0. "Setting (RD to logic 0) enables the CPU to read ADPCM data..."[OkiS90]. The controller then reads Port A of the PIA, which is connected to the speech chip data bus, and writes the data to the FIFO buffer at the location pointed to by Write_Ptr. Following this, 87_{16} = 1000 0111_2 is written to Port B, of which, bits B0 to B3 are connected to WR, RD, D/CB, and CS, respectively. This places RD to logic 1 and, thereby, completes reading of the speech chip. After data is read from the speech chip, the controller checks whether Write_Ptr is equal to Read_Ptr. If true, the controller interprets this as an implied stop command and jumps to the stop routine. If false, Write_Ptr is incremented to point to the next available location in the FIFO buffer, and the controller returns to the foreground

routine (RTI).

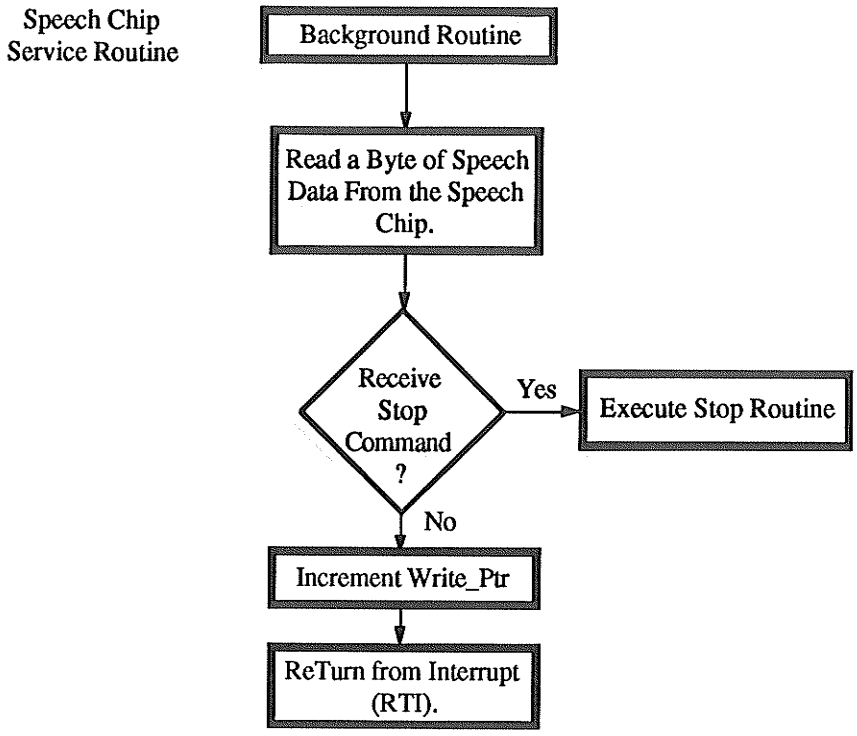


Fig. 4.27 Record background processing flow chart.

4.2.4.2.5 Stop

Upon detection of an implied stop command from either the playback or record routine, the controller branches to the playback stop code. The stop command is written to the speech chip by placing the command code on the data bus and then providing the sequence of appropriate logic levels on the input control signals WR, RD, D/CB, and CS. The stop command code, 01_{16}, is written to Port A of the PIA. This places the command code on the speech data bus. Following this, $82_{16} = 1000\ 0010_2$ is written to Port B, of which, bits B0 to B3 are connected to WR, RD, D/CB, and CS, respectively. This places WR to logic 0. About 12 μsec thereafter, $82_{16} = 1000\ 0011_2$ is written to Port B. This places WR to logic 1 and, thereby, writes the stop command to the speech chip. Also, the PIA interrupt relaying capability is disabled by writing $04_{16} = 0000\ 0100_2$ to CRA.

4.3 Serial Interface: RS–232C

This section describes the serial communications channel through which the memory manager and the host computer communicate (recall Fig. 4.18). In particular, the electrical, mechanical, and logical aspects of this channel are described. These aspects of the serial communications channel are compatible with the well known Recommended Standard 232 (RS-232C). In 1969 the Electronic Industries Association (EIA) issued the RS-232 interface. Since then revisions have been made, in particular, the RS-232C and RS-232D. While initially intended for the "Interface Between Data Terminal Equipment (DTE) and Data communications Equipment (DCE) "[CAMP 84], i.e., communications between terminals and modems, respectively, the RS-232C can also be used to interface computer and microcomputer serial communications, such as the application of interest. In order to fully specify the RS-232, four aspects are described:

- **Electrical Signal Characteristics** The voltage and logic levels of the serial data are defined.
- **Mechanical Connection Characteristic** The type, gender, and length of connectors and cables that form the physical link between systems are specified.
- **Functional Signal Description** Each wire that forms the physical link between systems is given a function, name, and corresponding pin number.
- **Standard System Configuration** Some system configurations are given to demonstrate the use of the RS-232 interface.

4.3.1 Electrical Signal Characteristics

The RS-232C interface specifies the binary logic levels and their associated voltage levels as shown in Fig. 4.28. The relationship is of the inverted logic type. That is, a logic

1 is represented as –15V, whereas, +15V is the representation of logic 0. Associated with the voltage range is the transition region. The transition region is the "dead band" area where the signal is undefined. Note that the threshold for output signals is 2V greater on either side of signal ground than the threshold for input signals. This difference is to allow noise margins and voltage drops along the length of cable. Thus, the dead band region for signals output into the channel is wider than that for input signals.

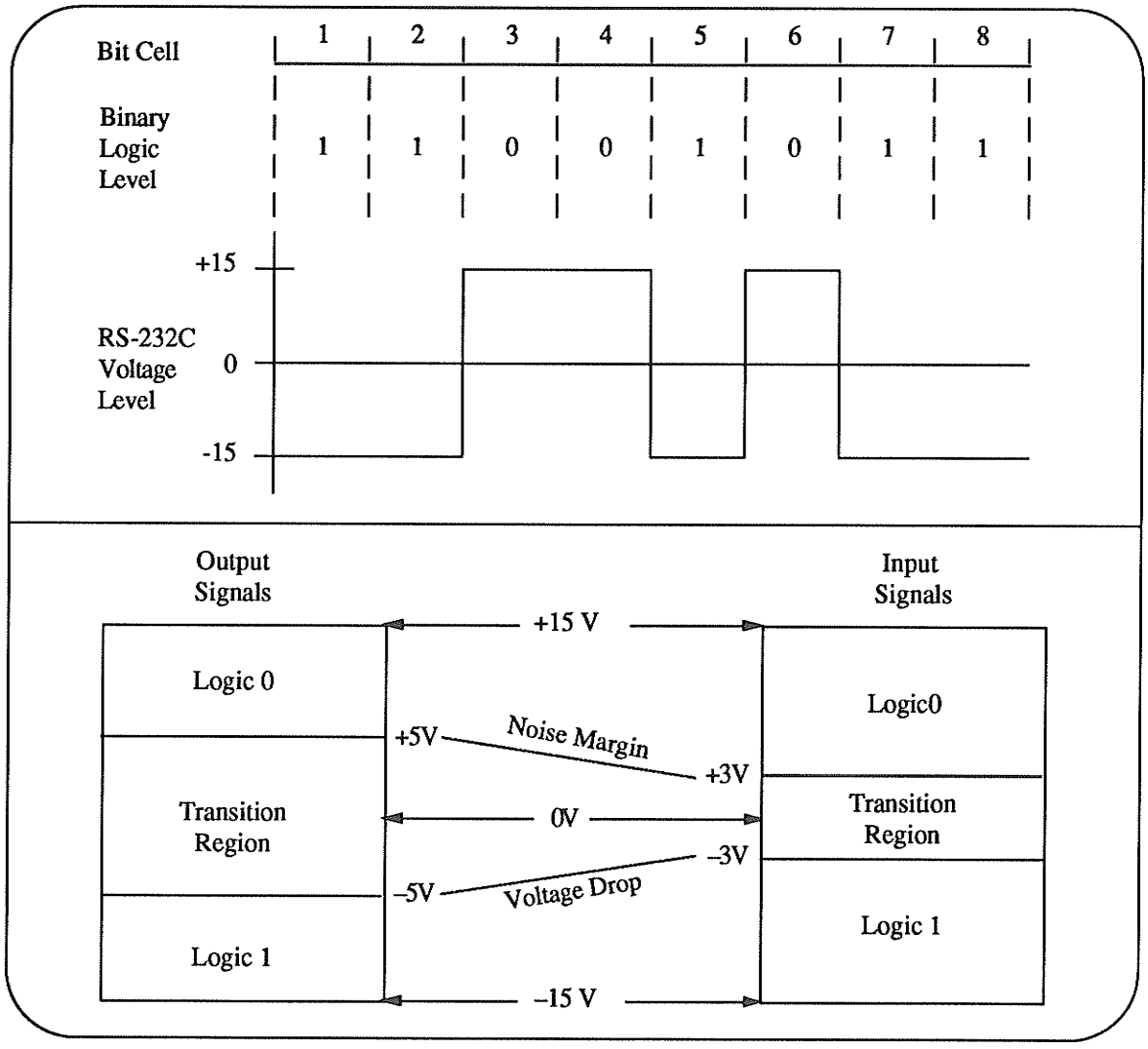


Fig. 4.28 RS-232C electrical signal characteristics.

4.3.2 Mechanical Connection Characteristics

The physical realization of the serial communications channel consists of a cable

terminated by connectors. The 25-pin D-shell connector is specified by the RS-232 interface. Thus, the physical medium has a capacity of carrying 25 wires, each of which has specific meaning and function, as described in Section 4.3.3. The maximum cable length is determined by the limit placed on the capacitance of the cable. This limit is 2500 pF, which means that, for an average value of 40-50 pF per foot, the maximum cable length is 50 ft. The type of cable is not specified, but telephone cable is usually used since modems transmit over telephone lines. This limit on cable length applies to transmission rates up to 20 kbps. However, by using shorter distances and better grade of cable, higher rates can be achieved. In particular, the serial communication channel of interest uses a cable length of 6 ft and transmits at 115.2 kbps.

4.3.3 Functional Pin Description

Figure 4.29 shows the functional pin assignment of the 25-Pin D-shell connector. A description of some relevant pins follows:

Chassis ground (CG) This signal is the protective system ground that links the ground signal of the DCE and DTE systems, i.e., the memory manager and the host computer. This signal is not the same as pin #7, the signal ground. The signal ground may be at a different potential than earth ground.

Transmit Data (TXD) Serially formatted data is transmitted on this pin. When no data is being transmitted, this pin is maintained at logical 1 or -15V.

Receive Data (RXD) Serial formatted data is received on this pin. This pin is also maintained at logical 1 when no data is being received.

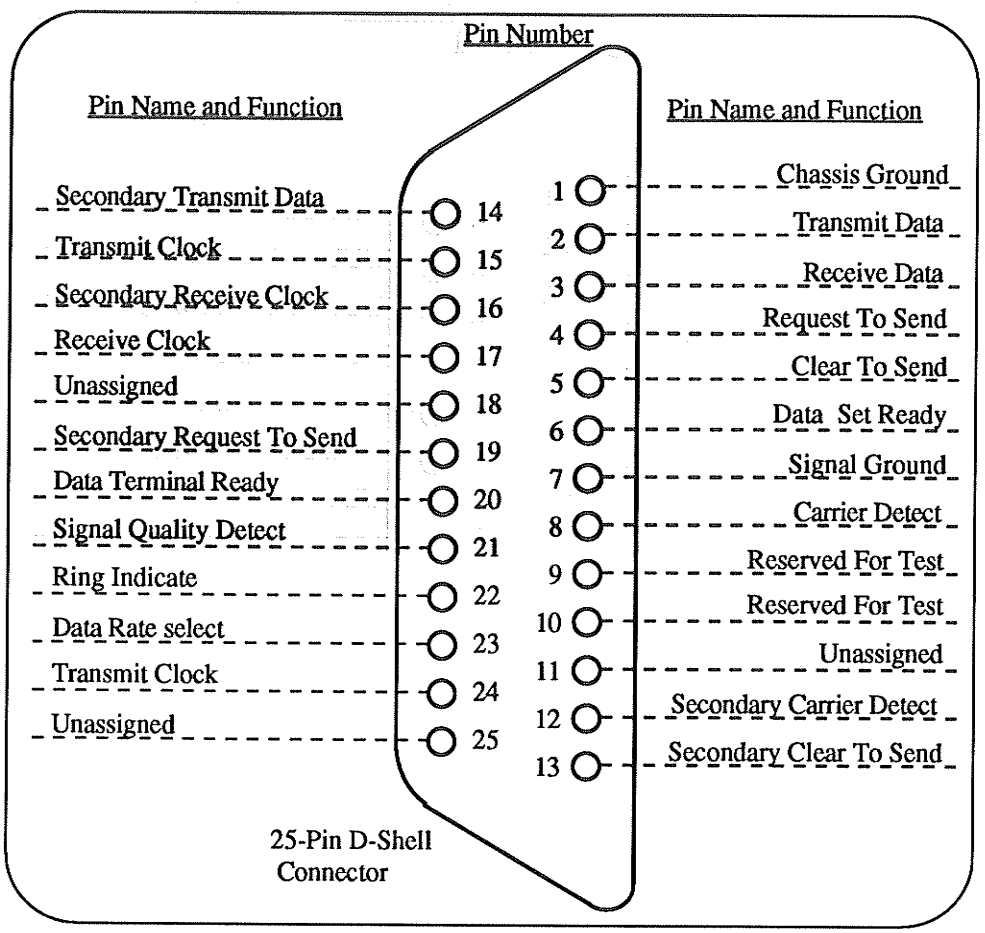


Fig. 4.29 RS-232C functional pin assignment [After JoCh87].

Request To Send (RTS) This is a control signal which, when active, sends a signal to the receiver requesting to send data. RTS is used in conjunction with Clear To Send (CTS) as a method of hardware handshaking to control data flow. The transmitter does not send data until it receives a clear to send by the receiver.

Clear To Send (CTS) In conjunction with RTS, this signal is used by the receiver and, when active, it tells the sender to start transmitting.

Data Set Ready (DSR) When this signal is asserted, a PC interprets that a modem is properly connected to the telephone line and in the data transmission mode.

Signal ground (SG) This is a mandatory signal ground. It defines the reference voltage for the data that is being transmitted or received.

Data Carrier detect (DCD) A modem sends a PC an ON signal when a proper carrier signal is being received.

Data Terminal Ready (DTR) Used as an output signal, DTR informs that it is powered up and ready to communicate.

The eight pins described above are the most important signals in the RS-232C interface, and they are appropriately called the BIG EIGHT. The remaining pins of the RS-232C interface are used for further hardware handshaking, backup, secondary duplicates of those already mentioned, or for test purposes. In the serial communication channel of interest, only 5 of the BIG EIGHT are used, since the remaining three are used mostly by modem communications. The host computer-memory manager system configuration which uses these five signals to form the serial channel is discussed next.

4.3.4 Standard System Configurations

Of the 25 signals offered by the RS-232C, the number and nature of signals used in the channel depends on the application. When interfacing a terminal to a modem, some or all of the BIG EIGHT are necessary, and this depends on the manufacturer of the modem or terminal [Camp89]. What is important is to determine what signals are required by the modem or terminal at both ends of the communication link. This can be done by referring to the manufacturer's data sheet specifications.

Interfacing computer to computer systems is relatively simpler. The simplest

method is referred to as a null modem. It is referred to as a null modem because the intermediate device, the modem, is eliminated. As shown in Fig. 4.30, only three signals are connected, TxD, RxD, and signal GND. Data is transmitted through wire TxD, and data is received through wire RxD. When, why, or how to transmit data is controlled through software handshaking.

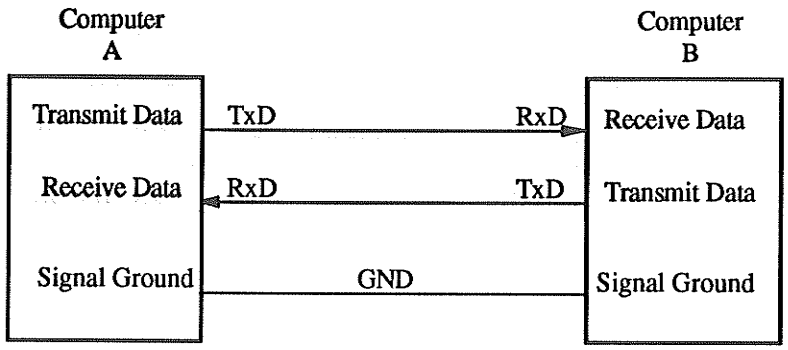


Fig. 4.30 Null modem configuration.

Software handshaking uses control characters to control the transfer of data between systems. One disadvantage of this system is that the control characters generally cannot be used as data. Most prevalent in software handshaking is the XON/XOFF protocol. XON and XOFF have several other names. XOFF is sometimes referred to as Device Control 3 (DC3), Ctrl-S, or 13_{16}, while the XON character is referred to as DC1, Ctrl-Q or 11_{16}. The receiver sends an XON when it is ready to receive data. Having received an XON, the transmitter begins sending data. If the receiver requires a break for any reason, then it transmits an XOFF. The transmitter ceases to transmit and waits for the next XON.

A more interesting system configuration – the interface used in the application of interest – is shown in Fig. 4.31. In this application six signals are used, TxD, RxD, RTS, CTS, DCD, and signal GND. This configuration enables hardware and software handshaking.

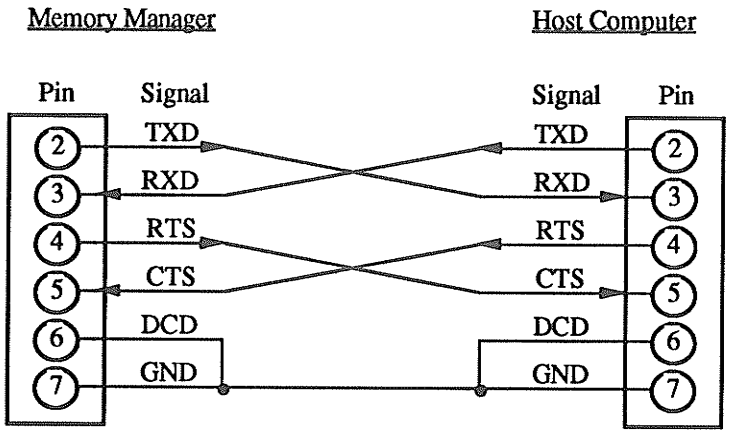


Fig. 4.31 Wiring diagram of host computer-memory manager interface.

The hardware method uses RTS and CTS to control data flow. This method is fastest of all handshaking protocols. If the transmitter wishes to transmit, it asserts output RTS. Because RTS of the transmitter is connected to the receiver's CTS, the receiver monitors its input CTS in order to determine when the transmitter requests to send. When CTS goes active, the receiver asserts its output RTS, which is connected to CTS of the transmitter. When the transmitter senses an active CTS, it begins transmitting through the TxD wire. The transmitter's TxD wire is connected to the receiver's RxD wire, and the receiver thus receives data through RxD.

4.4 Host Computer: the IBM or Compatible

The host computer used in this thesis is a Mind portable IBM compatible. This computer uses a 286 μP, 640 KB of RAM, a 40 MB Seagate hard disk, and a Microsoft mouse, version 7.0. It has a built in 7" by 5" screen driven by a Color Graphics Adapter (CGA) video card. It also has a RS-232C compatible serial port. All of the functions of the host computer, as discussed in Section 3.2.4, are implemented in software.

4.4.1 Software Description

Host computer software is written using 8086 assembly language, BIOS and DOS (version 3.0) interrupt calls [RaDu86], and Microsoft Quick C 2.0 [MSQC88] high level language. High level language is used primarily for the user interface. This includes setting up text and graphics user interfaces and main processing of user selected functions. Assembly language is used for basic processing of speech data. This includes configuring the serial port, receiving and transmitting commands and speech data, and converting data from one form to another. Whenever possible assembly language routines use BIOS and DOS system calls to communicate with PC hardware, such as keyboard and hard disk controller.

4.4.1.1 Text Mode Interface

The text mode of operation is used mainly to initialize the system, to record and playback speech, and to perform some library functions. Figure 4.32 shows the window which appears in text mode. The horizontal menu bar located at the top of the figure offers pull down menus and pop up dialogue boxes. Pressing the ALT key gives access to the main menu. Moving through the menu system is done by manipulating the arrow keys, while selecting an item is done by pressing Return. Typically, the first function performed is serial port initialization.

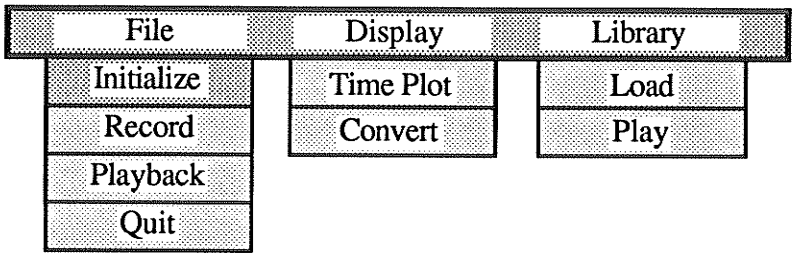


Fig. 4.32 Host's Main menu (text mode).

4.4.1.1.1 Serial Port Initialization

The serial port of the host computer uses the National Semiconductor 8250 UART [NaSe89]. The operation, function, and programming of this UART is very similar to that discussed in Section 4.2.1.2 for the ACIA. The UART's purpose is to provide the serial interface between the host computer and the RS-232C serial communications channel, as shown on the right hand side of Fig. 4.18. Initialization menu offers different data formatting, but the one used for speech data communications is as follows: 115.2 kbps, 8, N, 1. Data transmission rate is 115.2 kbps, characters are 8 bits in length, there is no parity bit, and one stop bit is appended to each character. Since Quick C, BIOS and DOS do not allow programming the serial port for transmission rates greater than 57.6 kbps, assembly language is used. The source code for initializing the serial port is shown in Appendix A.

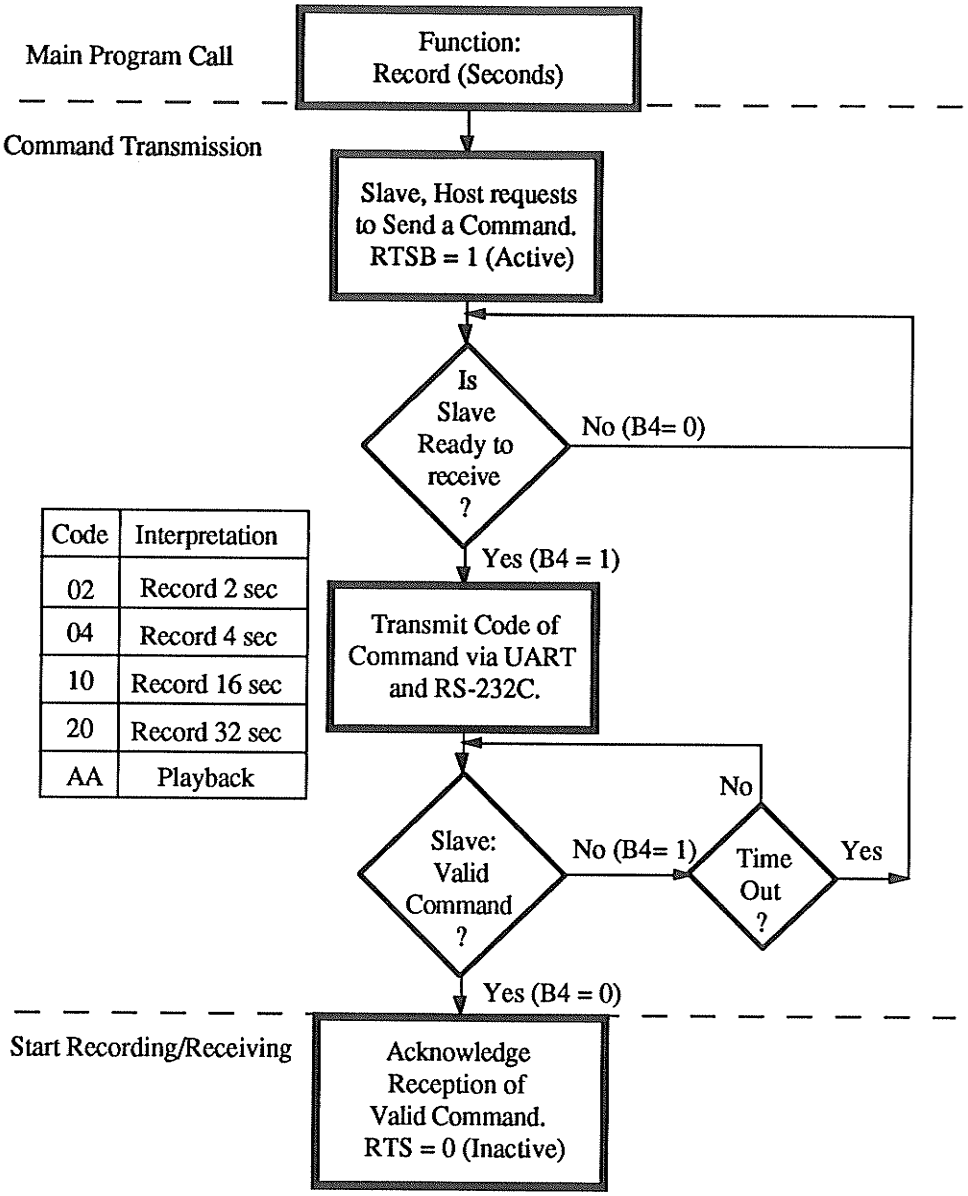
4.4.1.1.2 Record

Once the serial port is initialized as (115.2 kbps, 8, N, 1) recording can begin. Selecting record from the main menu, the user is prompted for the number of seconds to record. Currently, 2, 4, 16 and 32 seconds are offered, but this may be easily changed so long as the hard disk can accommodate the space. Since the recording rate is 4 kHz, i.e., 4000 bytes per second, the amount of space for recording t seconds is given as follows:

$$\text{Disk Space} = 4000 \text{ bytes/sec} \times t \text{ sec} = 4000t \text{ bytes} \tag{4.2}$$

After selecting the number of seconds, the main program passes the recording time to a record function, where two functions are executed, the first of which is SendRecordCommand. This function transmits the command code for "record t seconds" to the memory manager. Figure 4.33 shows the flow chart for transmitting a command, and this chart should be compared with Fig. 4.21, which represents the memory manager's

command reception flow chart. Note the mirror imaging of RTS and CTS, providing simple, yet effective, hardware handshaking.



Code	Interpretation
02	Record 2 sec
04	Record 4 sec
10	Record 16 sec
20	Record 32 sec
AA	Playback

Fig. 4.33 Send command flow chart.

When a valid record command is sent and received, the host executes the other function, record. Figure 4.33 shows a flow chart for the host's receive speech data routine. This figure is the mirror image of Fig. 4.25, and they should also be compared. As shown

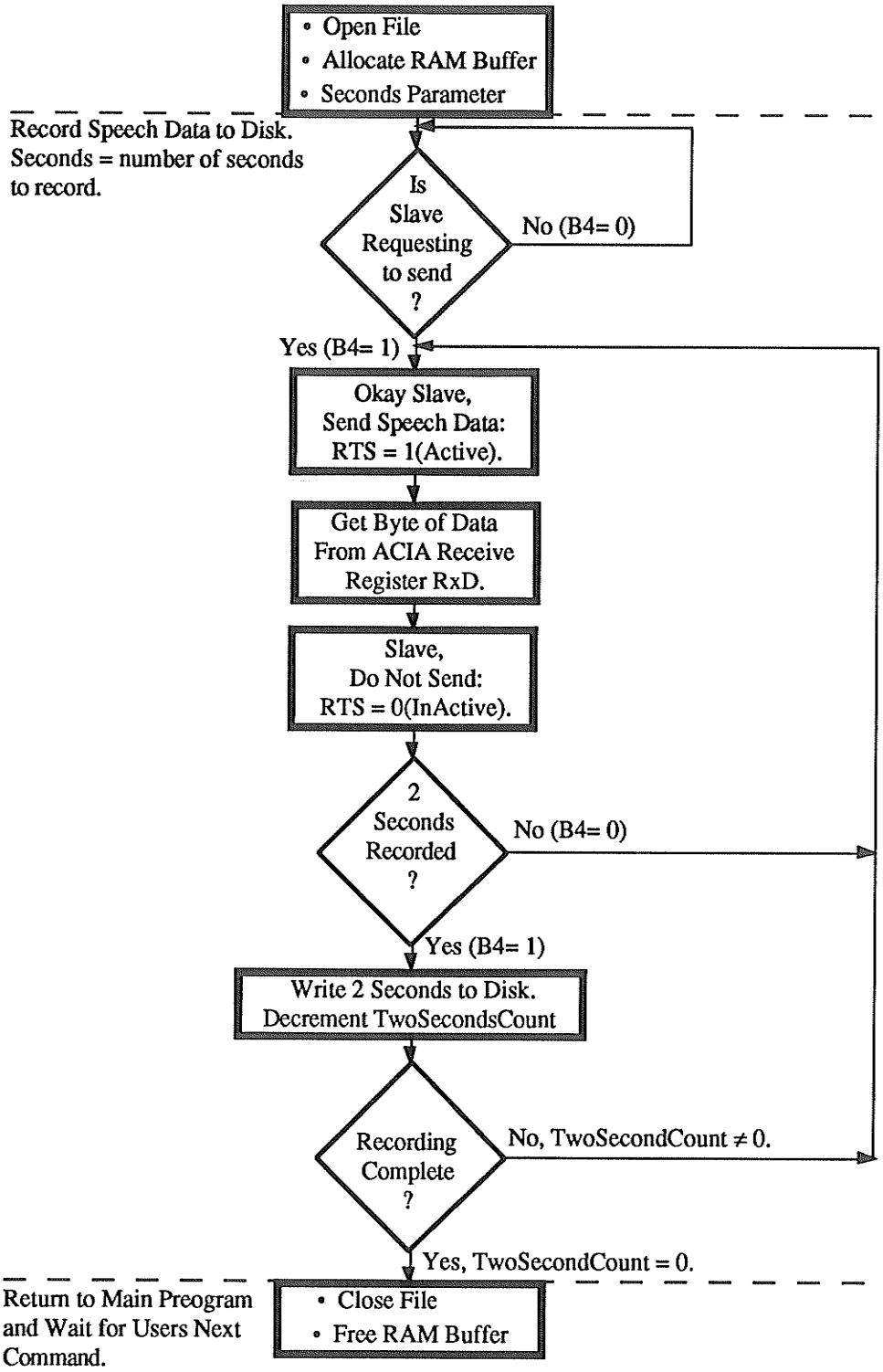


Fig. 4.34 Host receive speech data flow chart.

in Fig. 4.34, the record function begins by opening a file and allocating memory for a speech data buffer. This buffer holds 8192 bytes or approximately two seconds of speech

data. The Seconds variable is passed to the record function by the main program. TwoSecondCount counts each two seconds of speech. Before the host even attempts to start receiving data, it checks whether the slave is requesting to send, i.e., started the record process. Having determined slave's request to send, the host acknowledges the request by setting its RTS to logic 1, and waits for the incoming byte. Having received a byte, the host resets its RTS to logic zero in order to inform the slave not send. By manipulating RTS in this way, all 8192 bytes are received. Following the transmission of 8192 bytes, TwoSecCount is decremented. Note each decrement of TwoSecCount represents two seconds of recorded speech. If TwoSecCount decrements to zero, recording is over and the file is closed and the allocated RAM is freed. When finished, the host returns to its main program, where it waits for the user to select another command.

4.4.1.1.3 Playback

The process of transmitting, i.e., playing speech, is similar to that just described for recording. First, the playback command is transmitted. To start playback, the host begins by opening a file and dumping a portion of that file to RAM. The contents of this RAM are subsequently transmitted to the memory manager, using the same type of hardware handshaking as that described for the record mode.

4.4.1.1.4 Library Functions

Also available in text mode, speech recordings may be loaded into a library in order to form a template. Each entry in the library is stored in files. Furthermore, specific entries from the speech data library may be played back. The user can also remove selected speech files from the library. The purpose of forming a library is to facilitate speech editing tasks, which are possible in the graphics mode interface, as discussed in Section 4.4.1.2.

4.4.1.1.5 Compression

Also available in text mode is data compression and decompression routines. In order to plot speech data or directly modify the waveform, ADPCM formatted data must be converted to PCM. Similarly, in order to playback modified speech data, PCM formatted data must be converted to ADPCM. These conversion routines are based on the discussion of ADPCM presented in Chapter II, Section 2.2.3.1. The source code for these routines can be found in Appendix A, Section A.2.9.

Note that these routines implement a different algorithm than the one that is implemented by the manufacturer of the ADPCM speech processing chip, Oki (please see Section 4.5). In particular, the M values that are used as step size multipliers are not the same as those used by Oki (proprietary information). As a result, some information is lost in the conversion.

4.4.1.2 Graphics Mode Interface

In addition to having a text user interface, the host software also has a graphical interface. The graphical interface consists of initialization and time domain plot functions. The initialization function configures the other graphics software according to the type and capabilities of the current computer's video card. In so doing, the graphics support for this software should work with computers using CGA, EGA, HGC, MCGA, AND VGA, video adapter cards. However, when setting pixels in the time plot function, some minor modifications may be required for systems using cards other than CGA. In these cases, the incompatibility is partly due to screen resolution.

Normally a Graphical User Interface (GUI) product can produce a single interface

satisfying both text and graphics. However, text and graphics interfaces were written in order to limit the size of the program. For example, using a development toolkit such as Menuet v. 1.7d [CoLa90] to form the graphical interface requires a minimum of 200K of memory for just the overhead. The entire host program, including user interfaces and speech processing code, requires about 150K of memory.

4.4.1.2.1 Time Plot and Speech Editing

The graphical interface is used mainly as a speech processing tool. The main functions described in this section are time plot and speech editing. The graphical mode of operation is run when the user selects *Time Plot* from the main menu. Selecting *Time Plot*, causes a dialogue box to pop up, whereupon the user is prompted for the name and type of file to be plotted. Upon selecting a file, the file is plotted in a window similar to that shown in Fig. 4.35. Both ADPCM and PCM formatted files may be plotted, but PCM files are usually plotted, because they convey amplitude versus time information.

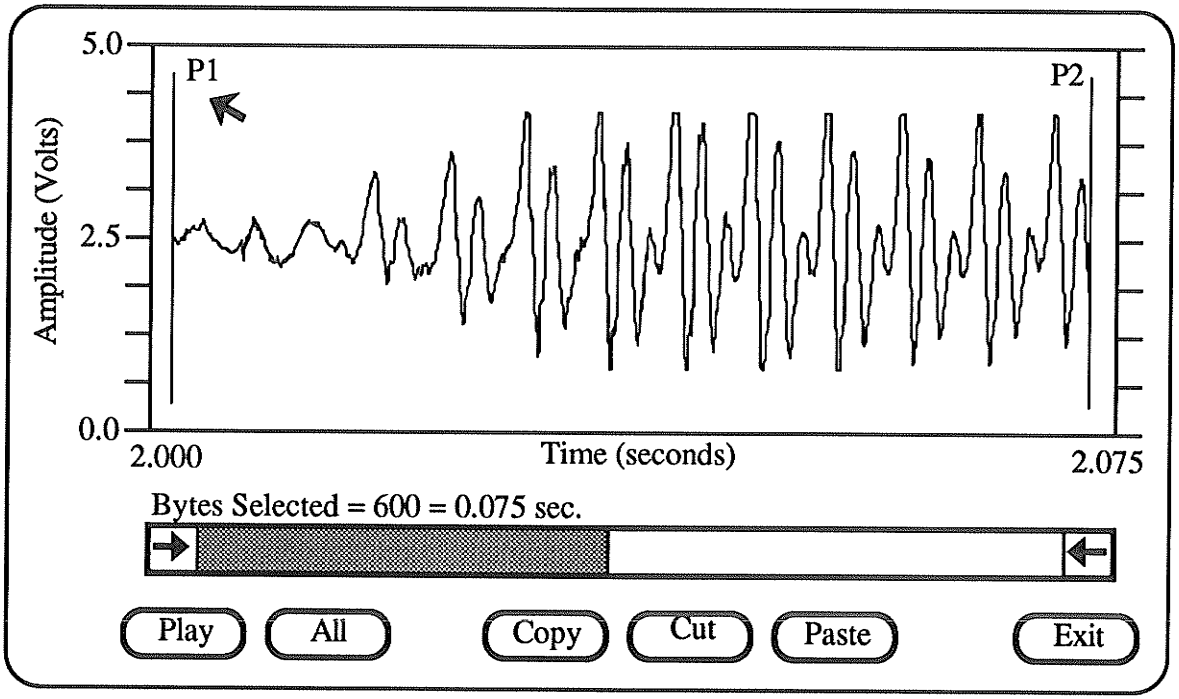


Fig. 4.35 Time plot window of host's graphical interface.

The horizontal bar near the bottom of the figure allows two ways of selecting a portion of a file to be plotted. The rectangular bar is scaled to the size of the file. Clicking the mouse inside the rectangle, causes 600 points to be plotted starting from the position of the mouse click. Finer movements within the file is provided by the arrows at both ends of the rectangle. Clicking on these arrows moves the file pointer 60 points (or 7.5 msec) in either direction.

There are two ways to select a portion of speech data. Clicking the *All* icon causes the entire file to be selected. Clicking anywhere within the plotting window causes up to two pointers to be displayed (see P1 and P2 in Fig. 4.35). It is possible to position the pointers anywhere within the file. One or both pointers may be removed by clicking on top of the respective pointer. Once the two pointers have been positioned, text underneath the plotting window indicates the number of bytes and corresponding time selected.

Playback is possible once a portion of speech is selected using the method described above. This allows the user to playback any portion of speech anywhere within the file. Also possible once speech is selected is *Cut* and *Copy* commands. *Cut* removes a portion of speech from the file, while *Copy* copies the selected speech to a clipboard file. The *Paste* command inserts speech data contained in the file clipboard into the current file starting at the location pointed to by one of the two pointers, P1 or P2. Note that only one pointer should be positioned for the *Paste* function.

Any size file may be viewed, selected, played, copied, cut, and pasted since the software uses the hard disk space as virtual RAM. In other words, rather than using RAM, the hard disk is used as a workspace. The disadvantage of this method is slower speed, that is because disk access time is considerably slower than RAM access. However, an immediate advantage is much more memory, and it is this that makes virtually unlimited size

of files for speech recordings possible.

4.5 Problems

Because of the unwillingness of the manufacturer (Oki) to disclose proprietary information about their ADPCM algorithm, some of the speech processing capabilities of the PC AT workstation could not be used. In order to plot speech data or directly modify the waveform, ADPCM formatted data must be converted to PCM. Similarly, in order to playback modified speech data, PCM formatted data must be converted to ADPCM. Because this cannot not be done exactly as performed by the speech processing chip, certain required speech processing functions cannot not be implemented on the PC AT workstation. In particular, any function dealing with modifying the amplitude of speech cannot be fully implemented, i.e., modifying the amplitude *and* playing back the result. These functions include amplitude interpolation and linear predictive extrapolation.

This problem is resolved quite simply by using an available Macintosh IIsi workstation for the functions not achievable on the IBM workstation. In fact, these two workstations are used in conjunction to conduct experiments, as discussed in Chapter VII Speech Splicing Experiments.

4.6 Summary

This chapter provides a detailed description of the PC AT speech processing system. No rock is left unturned. Everything you've always wanted to know about interfacing a 6802 μP, an MPU version of a speech processor, and a FIFO buffer to a host computer but were afraid to ask is not only discussed but explained in great detail.

The system consists of three main components, a speech processor chip, dual-

pointer FIFO buffer, and PC AT host computer. During record mode, the 6258 speech processor is responsible for digitizing the analog speech waveform and compressing the digital representation to an ADPCM format. During playback mode, the 6258 is responsible for decompressing from ADPCM to PCM and converting speech data from a digital representation to analog. For both modes, the speech chip provides *data ready* and *ready for data* signals in order that it be controlled by a microprocessor.

The dual-pointer FIFO buffer consists of a 6802 μP controller and SRAMs. The 6802 μP controls the speech chip and the communications between the speech processor and the host computer. During the record mode, the 6802 μP reads parallel speech data and writes these data to the SRAMs. In between reads, the 6802 μP transmits speech data from the SRAMs to the host computer via the serial communications channel. During the playback mode, the 6802 μP receives serially transmitted data from the host computer and writes these data to the SRAMs. In between receives, the 6802 μP reads parallel speech data from the SRAMs and writes these data to the speech chip. In this way, the FIFO buffer behaves as a data pipeline.

The FIFO buffer provides portability, isolation, and real-time disk capture and playing of speech data. However, because of the modular design of this buffer, it can be applied to other systems, not specifically for speech processing. This buffer essentially consists of an 8-bit parallel port and a serial port. As such, any parallel organized system requiring asynchronous communications with a serial organized system may use this buffer. The current limitations are that the serial organized system can transmit not greater than 115.2 kbps, and the parallel system must transmit less than one-half of 115.2 kbps, i.e., approximately, 5.5 kHz.

Most of the speech processing is done on the host computer. The host computer performs real-time disk capture, thus recording and playing time is limited only by the space

available on hard disk. A serial port initialization routine is included. ADPCM formatted data is converted to PCM format and vice, versa (note that this function is currently unavailable due to reasons as discussed in Section 4.5). In the time domain plot function, the time domain plot of PCM formatted data is displayed. In this window, any portion of speech data located anywhere within a file may be selected for processing. This processing includes, playback, copy, cut, paste, linear predictive extrapolation, and averaging. Any size file prefixed by 'PCM' may be processed since this software uses the hard disk as virtual RAM.

The dual-pointer FIFO buffer is not the only buffer capable of performing the job for a speech processing system. The next chapter provides a paper design of an alternative buffer, a swinging buffer implemented in hardware.

CHAPTER V
ALTERNATIVE BUFFER DESIGN

The previous chapter discusses a microprocessor based design of a controller for a speech data buffering system. The microprocessor or microcontroller approach is, generally, preferred because software provides flexibility. By using a microprocessor instruction set and an evaluation board to do the debugging, the designer acquires tolerance for the logical, technical, or wiring errors that are bound to occur (by Murphy's Law). Many of the problems encountered in the design of a software based controller are solved by modifications to the software; no hardware changes are required.

However, the microprocessor based controller may not yield the most efficient design in terms of optimizing the chip's capability and speed. For example, in the design of the speech data buffer of interest, the full potential of the microprocessor is not realized. The controller requires a small subset of the instruction set, such as 'move data from the PIA to Buffer A' and 'compare memory addresses'. Also, the microprocessor requires time to execute instructions and to respond to interrupts. Rather than responding to an event as soon as it occurs, the microprocessor response is delayed (μsec delay). In this way, the microprocessor response time is quantized. Consequently, accessing data at a particular instant may not be possible with a microprocessor. Nevertheless, the knowledge gained by the microprocessor based design approach, gives the designer insight into an equivalent hardware based design of a controller.

This chapter presents an alternative solution to the design of a controller for a speech data buffering system. Rather than using a microprocessor, the controller is designed using a digital circuit consisting of primitive logic gates (the AND, OR, and

INVERTER) whose combinatorial logic is capable of controlling the communication of control signals and speech data. The hardware circuit design approach is a refinement of the microprocessor based design. This paper design is intended to be implemented using the Xilinx Logic Cell Array (LCA) technology.

5.1 System Design And Description

The apparatus of the speech processing system for the hardware controller design approach is shown in Fig. 5.1. Notice that, unlike the previously presented system (as shown in Fig. 3.1), the only external hardware is the microphone. All other necessary circuits are contained within the host computer, which is the Macintosh II. While this design has its disadvantages (such as the isolation problem mentioned in Chapter III, Section 3.3.3), there are obvious advantages, such as, simplicity and compactness. For this system all that is necessary for operation is connecting the microphone to the back of the Macintosh and then starting the Macintosh speech processing system menu driven software.

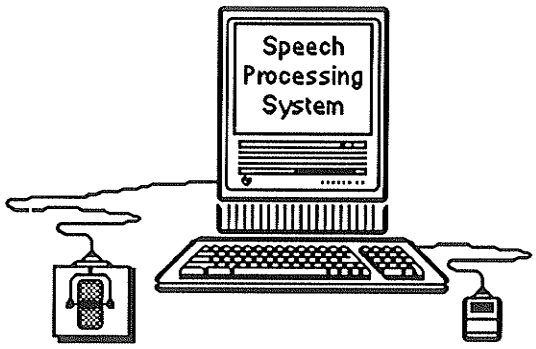


Fig. 5.1 Speech processing system.

The previous figure shows an external view of the system. To get an idea of how such a system may work, let us take a look at an internal view as shown in Fig. 5.2. Internally, the Macintosh II mainly consists of the 68030 microprocessor, four ROM chips, up to 8 MB of SIMM RAM memory, a serial communications controller for the modem

and the printer, an SCSI parallel port typically used for a hard disk, built in speaker and amplifier, and as well as many other nice features. As can be seen in the figure, the speech data buffer board is added to an available expansion slot within the Macintosh. What this connection immediately implies is that the buffer board shares the external data, address, and control buses with the host CPU, the 68030.

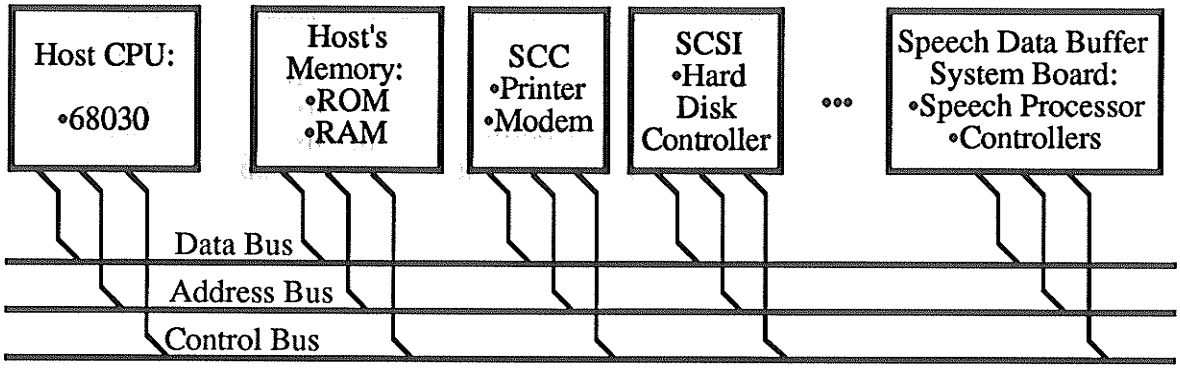


Fig. 5.2 System block diagram. The speech processor and several controller circuits are connected together on one PC board which is added to one of the available expansion slots of the Macintosh.

With this connection in mind, we can see how the speech processing system works. For example, to get the buffer to perform a certain function, such as record or playback, the host computer first writes the command to the buffer. The code of this command is placed on the 68030 external data bus, and the address of the buffer's command register, that will accept and process this command, is placed on the address bus. Proper address decoding ensures that the buffer and only the buffer receives this command. The part played by certain control signals on the control bus is to latch the command, which is on the external data bus, into the buffer's internal command register. Once the command register is loaded, processing of the command begins.

During the record mode, the host processor, the 68030, recieves data from the buffer board as follows. The host computer must first address and read the buffer's internal status register, which indicates whether data is ready. If data is ready, the host addresses the

buffer's internal data register, which contains fresh data. The data is placed on the external data bus and eventually makes its way to the host's private RAM memory or to the hard disk for non-volatile storage, to be processed at a later time. The host computer continues this way until it is decided to terminate the record mode, in which case, the host merely addresses the command register and writes the record stop code.

The procedure for playback is similar. The code for the playback mode is written to the command register and this is followed by the commence playback command. The host computer then reads a status bit in the status register that indicates whether the speech chip is ready to playback a byte of speech data. If so, the host writes a byte of speech data to the data register. The host computer continues this way until it is decided to terminate the playback mode, in which case, the host merely writes the playback stop code to the command register.

5.2 Speech Data Buffer Board

To better understand how the buffer board performs the above functions, let us take a closer and more detailed look at the block diagram of the speech data buffer as shown in Fig. 5.3. The board consists of the speech processor and the data buffer controller, which, in turn, consists of the interface controller, the read and write timing controller, and the swinging buffer controller. The interface controller is responsible for intermediating between the physical and logical aspects of the communication between the host computer and the buffer board. The memory manager controller is also an intermediary device, which arbitrates between the different data rates of the speech processing chip and the host CPU. The read/write controller is responsible for supplying the speech chip and the memory manager controller with data access timing signals. The most immediate circuit facing the host CPU is the interface controller.

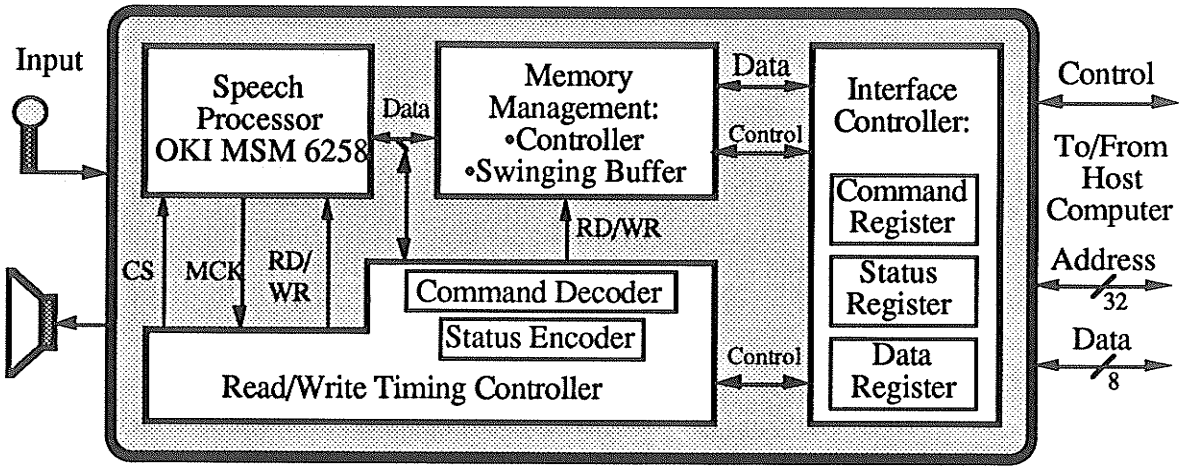


Fig. 5.3 Layout of the speech data buffer board.

5.2.1 Interface Controller

There are two conceptual functions of the interface controller. The memory circuits placed at the physical boundary and the protocol prescribed at the logical boundary between the host's external bus and the buffer's I/O port constitute what is called the interface controller. Conversely, the command register, the status register, and the data register of the interface controller act as the physical interface, separating the external bus of the Mac II and the internal bus of the buffer. On the other hand, the logic of the codes used for commands and status and the method by which these codes are transmitted, received, and decoded serve as the logical interface.

The IEEE standard NuBus interface is the protocol adopted and used by the Mac II. [Appl87] is a good reference for designing cards for the Macintosh. The NuBus protocol is a sophisticated extension of the familiar memory mapped concept of the 68000 family microcomputers. What this means is that cards designed for the Macintosh should follow the NuBus or at least the memory mapped protocol.

The buffer device is a memory mapped system and is accessed broadly similar to

memory transfers, and the buffer is said to occupy part of the 68030's address space. In particular, the board is assigned three unique addresses, one for data, one for command, and the other for status information. Only eight bits of these locations are used by the buffer because the speech processor is an eight bit machine, i.e., the 6258 speech processor has an eight bit external multiplexed data/command/status bus.

The buffer interface monitors the Macintosh's address lines and detects when the buffer board is being addressed. When properly addressed, the controller latches command, status, or data onto the 68030 external bus or into the appropriate internal registers. The controller also enables other buffer controllers according to the specific command in the command register. The interface controller holds the buffer board in a disabled state when not addressed.

In order to communicate properly, the codes of the commands issued by the host and the status supplied by the buffer are defined. Table 5.1 shows the meaning of each code.

Table 5.1. Command codes.

Command Register								Interpretation
C7	C6	C5	C4	C3	C2	C1	C0	
X	0	1	0	X	1	0	1	Record Command: Initialize
X	X	X	X	X	0	1	0	Start Record
X	0	0	1	X	1	0	1	Playback Command: Initialize
X	X	X	X	X	0	0	0	Start Playback

C0 is Command/ Status Trigger	C1 = 1 RD 0 WR	C2 = 1 Data 0 Command	C4 C5 C6 = Command word

5.2.1.1 Circuit Description

As shown in Fig. 5.4, the interface controller circuit consists of an address decoder, an 8 bit command register, an 8 bit status register, and an 8 bit data register.

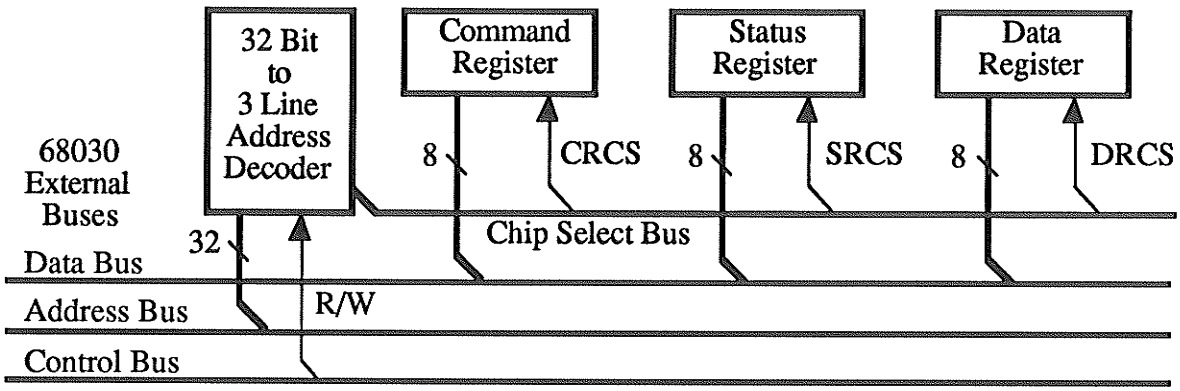


Fig. 5.4 Interface Controller.

The address decoder monitors the 32 bit address bus and asserts either the Data Register Chip Select (DRSC), the Command Register Chip Select (CRCS), or the Status Register Chip Select (SRCS) if and only if their corresponding address is on the bus. Afterwards, and if the corresponding register has been enabled, the interface controller uses the 68030 issued read/write signal to latch the data, command, or status byte either onto the external bus or into the respective register. Furthermore, upon a valid address, an enable control signal is sent to other controllers and associated circuits of the buffer board. Otherwise, when the buffer is not being addressed, all registers and the other circuits of the buffer are disabled.

When a command is latched into the command register, the control bits of this register are used as inputs by other circuits of the buffer in order to execute the command. One of these circuits is the read/write timing controller.

5.2.2 Read/Write Timing Controller

The read/write timing controller circuit consists of two major parts, the read/write timing signal generator circuit and the command decoder and status encoder. The read and write signal circuit is responsible for generating Chip Select (CS), ReaD (RD), and WRite (WR) timing signals as required by the speech processor and in part by the swinging buffer of the memory manager controller. The command decoder is used to decode commands issued by the host computer and to present the commands to the speech processor. The status encoder is used to encode the status information read from the speech processor and to present the status to the host computer. The read/write signal generator is discussed first. This is followed by a description of the command decoder and status encoder.

Because the signals of the read/write circuit are used for data access, their timing is critical, and a thorough understanding of their purpose and their relative timing is important. Furthermore, the design of the circuit must conform to the prespecified timing diagrams. Accordingly, the description of the circuit design is preceeded by a description of the timing diagram, which is supplied by the manufacturer, Oki, of the speech processing chip. The write timing is similar to the read timing, and, therefore, a detailed write timing description is not given, although some of the differences are discussed.

5.2.2.1 Timing Description

Fig. 5.5 shows typical waveforms and the timing of signals required for reading of speech data and status information from the speech chip during the record mode.

In order to communicate with peripheral devices, the speech chip offers some output control signals. The Voltage sampling ClocK (VCK), the sampling frequency, and the

Microprocessor ClocK (MCK), the data ready flag, are output control signals intended so that a peripheral device may synchronize itself with certain events taking place within the speech processor. The falling edge of VCK indicates that the speech processor is beginning to convert an analog speech sample to digital form. Subsequent samples are converted every 125 μsec (i.e., 8 kHz). The falling edge of MCK, which occurs every 250 μsec (i.e., 4 kHz), indicates that a pair of ADPCM nibbles are 'ready' for reading. Note that MCK does not occur at a fixed time in respect with VCK. It is for this reason that the read/write timing controller derives its timing with respect to MCK, rather than VCK.

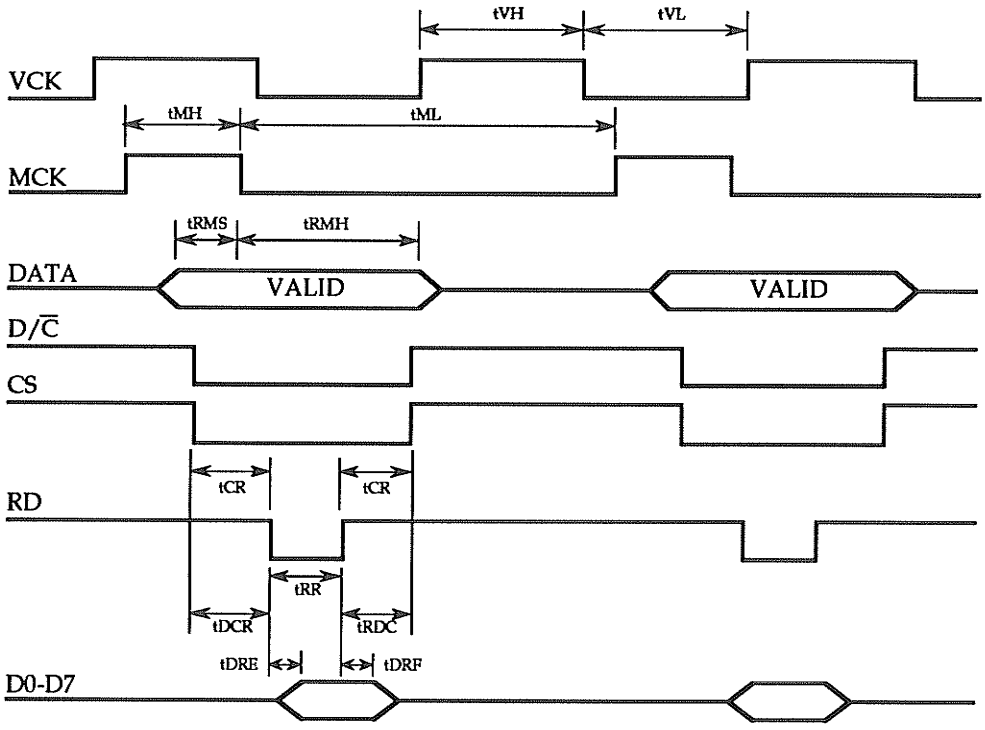


Fig. 5.5 Speech processor read and status output timing diagram.

In order to complete the communication link, the speech chip expects some input control signals. The CS, RD, and D/C are input signals required by the speech chip for the record mode. CS is used to enable the chip, D/C is used to select the speech data mode or the command/status mode, and RD is used to read speech data or status information from the speech chip.

The timing diagram specifies a valid data access window. The time tRMS + tRMH = 70 μsec is the time during which internal data is valid. However, before valid data can be accessed at the external pins of the chip, the chip requires a setup time, tRMS = 15 μsec. The chip indicates when this time, tRMS, has elapsed by pulling MCK low. Following the negative edge of MCK, the chip allots a tRMH = 55 μsec window (hold time) for an external device to present the CS, D/C, and RD signals and thereby access data or status. Note that the presentation of RD must be within the chip enable window created by CS, i.e., data is actually latched at the positive edge of RD.

Following the data ready signal given by MCK, the D/C, CS, and RD signals are presented within the tRMH window. Upon the negative edge of MCK, the speech chip is selected and the data mode is chosen through the assertion of CS and D/C, respectively. Note that CS and D/C may occur at the same time. At least tCR = 50 nsec thereafter, the RD signal is asserted causing internal circuits to begin latching speech data onto the external pins. However, before an external device can read valid data, the chip requires tDRE = 200 nsec in order to stabilize the data onto the pins of the chip.

After asserting the read data signals, the external device must withdraw them. After the assertion of the RD signal, a delay of at least tRR = 250 nsec is recommended before RD is disasserted. The positive edge of RD at this moment is actually when data is latched and read by the external device. After at least tCR = 50 nsec from the raised edge of RD, the CS signal is disasserted and D/C may also be withdrawn. The external device then waits for the next negative edge of MCK, whereupon the same sequence of events continues. Fig. 5.6 shows a flow diagram of the sequence of events described above.

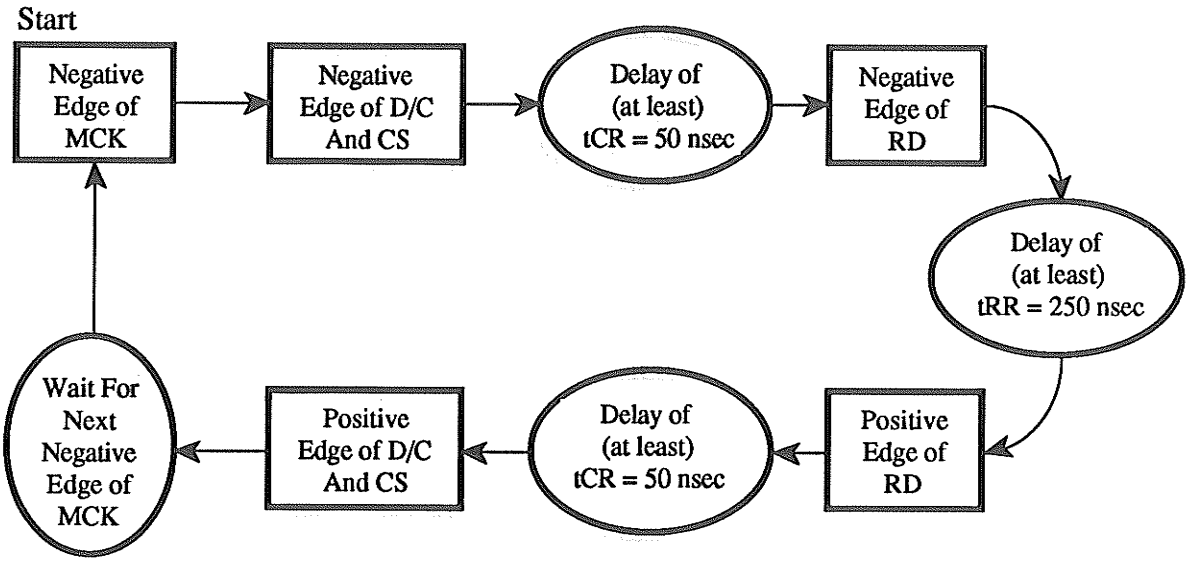


Fig. 5.6 Read signal timing and presentation. Sequence of events required for the correct timing and generation of CS, D/C, and RD signals.

The timing for writing commands or data to the speech chip is similar, one difference is that the data ready signal is signified by the positive edge of MCK, rather than the negative edge.

5.2.2.2 Circuit Description

A circuit implementation of the timing specifications described above is shown in Fig. 5.7. As shown in this figure, the circuit consists of a Flip Flop (FF), a 3-bit binary

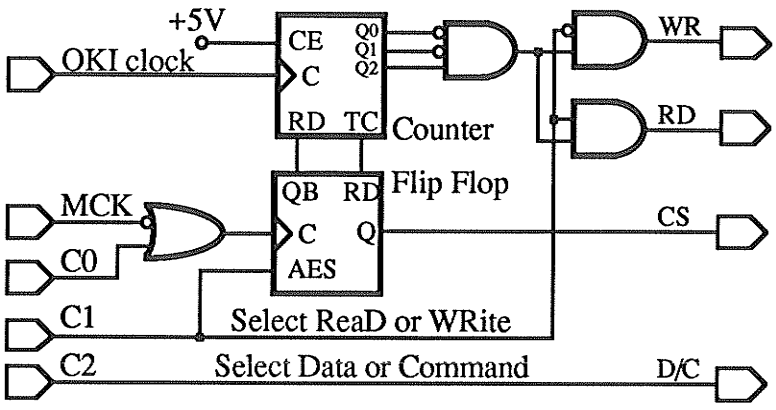


Fig. 5.7 Schematic diagram of the read/write timing controller circuit.

counter, and some primitive logic gates. The inputs to the circuit include the MCK data ready signal and the C0, C1, and C2 command bits, while the outputs are CS, D/C, RD, and WR signals.

The circuit can be in one of five modes: active and generating signals for command write; active and generating signals for status read; active and generating signals for record mode; active and generating signals for playback mode; and inactive. The circuit is activated by either active edge of MCK or C0 and is inactive by their absence. Typically, C0 activates the circuit first in order to write a record or playback command to the speech chip. When activated by C0 for a command write, the appropriate timing signals, i.e., CS, D/C, and WR, for one write to the speech processor are generated. MCK is used to activate the circuit during the record or playback modes. For every active edge of MCK, the appropriate timing signals are generated for one read or for one write of speech data.

The purpose of the FF is relay active edges of MCK or C0 to other parts of the circuit. The active edge is selected by the command bit C1, which is connected to the Active Edge Select (AES) pin of the FF. The FF is positive edge triggered for writing commands or data to the speech chip and negative edge triggered for reading status or data from the speech chip. The FF also holds the counter in its reset state when the circuit is inactive.

The purpose of the counter is to provide the CS duty cycle and the RD or WR duty cycle within the CS window as specified by the timing diagram described above. When the circuit is idle, the counter is held at a reset state, since the reset input of the counter is connected to the inverted Q output of the FF (QB is high on reset). When the FF triggers, Q goes high (the FF is self toggling) and, thus, the counter begins counting. The raising of Q provides the first edge of CS, since Q is tied the CS input of the speech proccessor. The Oki clock is used to provide the rate at which the counter counts and to synchronize the

counter with the speech processor.

A combination of the outputs of the counter provide the RD or WR signal. As shown in Fig. 5.8, the logical combination, (NOTQ0 AND NOTQ1 AND Q2), is chosen so that the duration of the asserted RD or WR signal exists within the assertion of CS. When the counter reaches its maximum count (i.e., seven), the FF is reset and, consequently, Q goes low. The lowering of Q provides the falling edge of CS, and thus the CS window is manifested. As a result of QB = 1, the counter is once again held at a reset state. The counter then waits for the next stimulation of the FF.

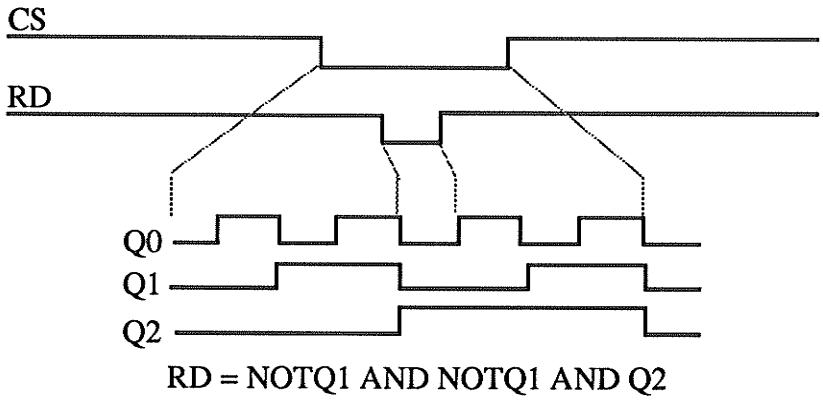


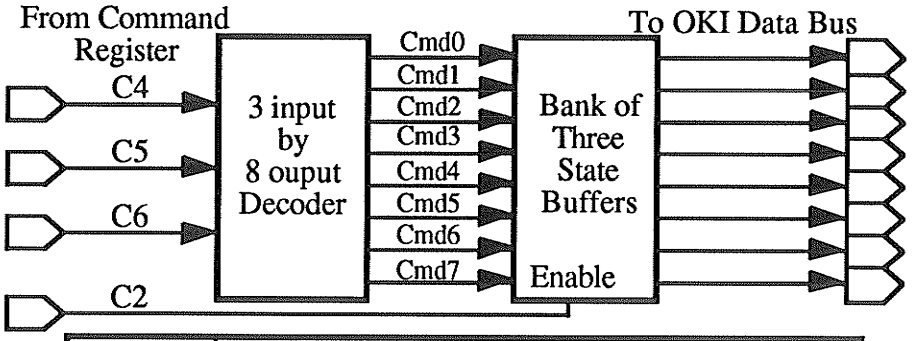
Fig. 5.8 Timing of CS and RD or WR signals. CS is asserted throughout the duration of the count (0, 1, 2, 3, 4, 5, 6, 7). The RD (or WR) signal is actually the fourth count of the counter. Thus, RD (or WR) exists in its asserted state within the CS window.

Section 5.1 mentions that in order to start recording or playback, the host computer must first write a command to the buffer board. Section 5.2.1 went on further to explain that the interface controller actually stores the command in the command register. The purpose of storing the command is so that other circuits of the buffer can use the command to initialize properly, and hence, to begin the process indicated by the command. Furthermore, section 5.2.2.2 discusses how the control bits C0, C1, and C2, of the command register, are used for initialization and process control.

Discussed next is the purpose of the other control bits of the command register. In particular, it is shown how the control bits C4, C5, and C6 are used to decode the command, and, hence, initialize the process indicated.

5.2.2.3 Command Decoder

The purpose of the command decoder is to interpret the command sent by the host computer and to inform the speech processor of what process is to be taken. The circuit responsible for interpreting and relaying commands is shown in Fig. 5.9. The circuit simply consists of a 3 by 8 decoder and a bank of three state buffers. Bits C4, C5, and C6 represent the binary code of the command. The decoder decodes the binary code to an 8-bit representation recognizable by the speech chip. The reason for using three state buffers is that the command bus must be in the high impedance state when the speech processsor is in the record or playback mode, otherwise there will be bus contention. Note that C2 is used for two related purposes. If C2 is high, the data mode of the speech processor is chosen and the buffers are placed in high impedance. If C2 is low, the command mode is selected and buffers are enabled.



Binary Input	Decoded Command	Speech Processor Function
C6 C5 C4	Cmd7 ... Cmd0	
0 0 0	0 0 0 0 0 0 0 0	Not Defined
0 0 1	0 0 0 0 0 0 0 1	Stop
0 1 0	0 0 0 0 0 0 1 0	Play
0 1 1	0 0 0 0 0 1 0 0	Record

Fig. 5.9 Command decoder circuit. This circuit decodes the binary code of the command to a format recognizable by the speech processor. The high impedance state of the buffers (i.e., if C2 = 1) ensures no bus contention.

Consider an example of how to start recording speech. The reset code and the initial state of the command register of the interface controller is 00000000. Before recording can start, the host computer must write two commands to the command register. Referring to Table 5.1, the first code is for record initialization, X010X101. Bits C4, C5, and C6 are input to the decoder as shown above. The decoder's interpretation is 00000010. Since C2 is high, the three state buffers are enabled and the record command 00000010 is placed on the speech processor's data bus. Referring to Fig. 5.7, because C0 went from low to high, the FF is triggered and subsequent CS and WR signals are generated (note WR is generated instead of RD because C1 is low). The presentation of CS and WR causes the data appearing on the data bus, 00000010, to be latched into internal registers of the speech chip. The speech chip now begins recording. Every 250 μsec fresh speech data will be available on the data bus. However, in order to prevent bus contention, the previous command must be taken off the bus. This is achieved indirectly through requiring that the host issue another command. The next code that the host computer writes is XXXXX010, which, from the host's point of view, means start. From the buffer's point of view, this code is required in order to take the command code off the speech data bus and reinitialize the control signals for the record mode. Because C2 is low, the three state buffers are placed in high impedance, and, therefore, the command is taken off the bus. C1 is set high because the read/write controller is required to periodically generate the RD pulse for the record mode. C0 is reset to low in order to allow MCK to take over the job of triggering the FF. Also C0 is reset low in anticipation for the next command issued by the host computer, such as, stop.

Once the record command is decoded and written to the speech processor, recording of speech starts automatically. Thereafter, as indicated by MCK, new data is ready every 250 μsec, and it is expected that the host computer reads each byte of data on time so that no data is lost. This imposes an inconvenient and synchronous constraint on the host

computer. While perfectly able to communicate data transfers with the speech processor at its relatively slow rate of 4 kHz (i.e., every 250 μsec), the host computer prefers a much higher rate. The host, in fact, requires a higher rate of transmission because, in order to record and playback speech in real time, a certain amount of time is required to save speech data to non volatile disk, without stopping the recording or playing process. Because the host requires more than 250 μsec to save a block of data to disk, the host requires to communicate data transfers in bursts rather than one byte at a time. This necessitates some sort of buffering technique to be employed between data transfers of the speech processor and the host computer.

5.2.3 Memory Manager

The purpose of the memory manager is to alleviate the synchronous nature of data transfers between the speech processor and the host computer. To fulfill its purpose, the memory manager employs the concept of the swinging buffer. The swinging buffer is a two port device which allows relatively slow data transfers at one port, while allowing much faster transfers at its other port. The swinging buffer technique of memory management is particularly suited for the speech processing system of interest since the speech processor communicates data at 4000 bytes per second (Hz), or 32 000 bits per second (32 kbps), while the host computer is capable of megabytes per second, which is the speed of the host CPU, the 68030.

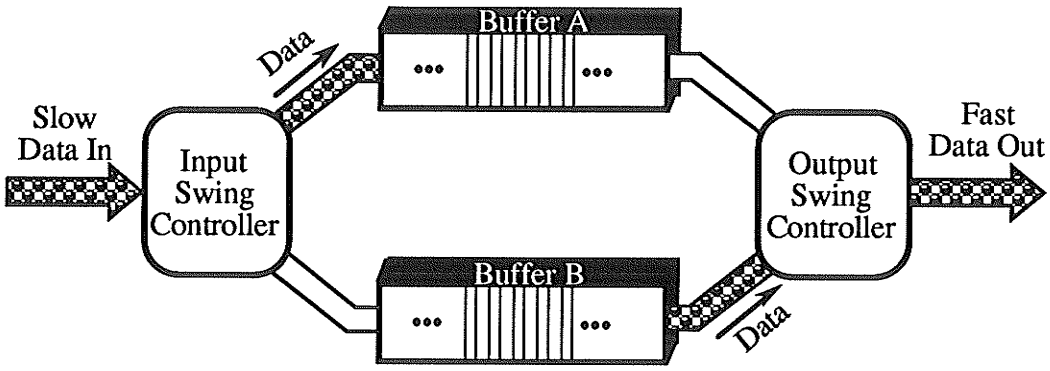


Fig. 5.10 Swinging buffer block diagram (after [Kins88]).

As shown in Fig. 5.10, the memory manager consists of an input and output swing controller and two buffers, which, in this application, are implemented as First-In-First-Out (FIFOs) memories.

The swing controller, as its name implies, controls the path switching mechanism and determines which buffer is connected to which device at any given time. In order to prevent instances when two devices try to access the same buffer at the same time, only one buffer is permitted to be connected to a device at a particular moment. For example, while

buffer A is connected to the speech processor, buffer B is connected to the host computer, and vice versa. Note that the host computer is effectively isolated from the speech processor, and this is what enables the host computer to asynchronously read speech data at a much higher rate than that transmitted by the speech processor. When the host computer eventually empties buffer B (and it will empty buffer B at a fraction of the time required by the speech processor to fill buffer A), the host stops its receiving routine and starts saving speech data to disk. When the host computer finishes saving data, it returns to its receive routine and waits for the next buffer full signal. In the mean time, the speech processor continues filling buffer A. When buffer A becomes full, a buffer full signal occurs and the speech processor is connected to buffer B (swing to buffer B) and the host computer to buffer A (swing to buffer A).

The swing controller not only controls the path switching mechanism but also derives data access signals, i.e., the chip select signals that are actually responsible for reading from and writing to the buffers. The buffer chip select signals are derived from RD and WR signals and CS signals associated with speech processor and host computer reads and writes, respectively. For example, during playback, CS signals generated by host computer writes to the speech data buffer are directed to one buffer, say buffer A, while WR signals generated by the read/write controller for buffer reads and speech processor writes are directed to the other buffer, buffer B.

5.2.3.1 Swinging Buffer Timing

In order to give a clearer picture of how the swing controller determines which buffer should be connected to which device at any given time and to which buffer the RD, WR, and CS signals should be directed, a timing diagram is given. Fig. 5.11 shows the timing diagram of the swing controller.

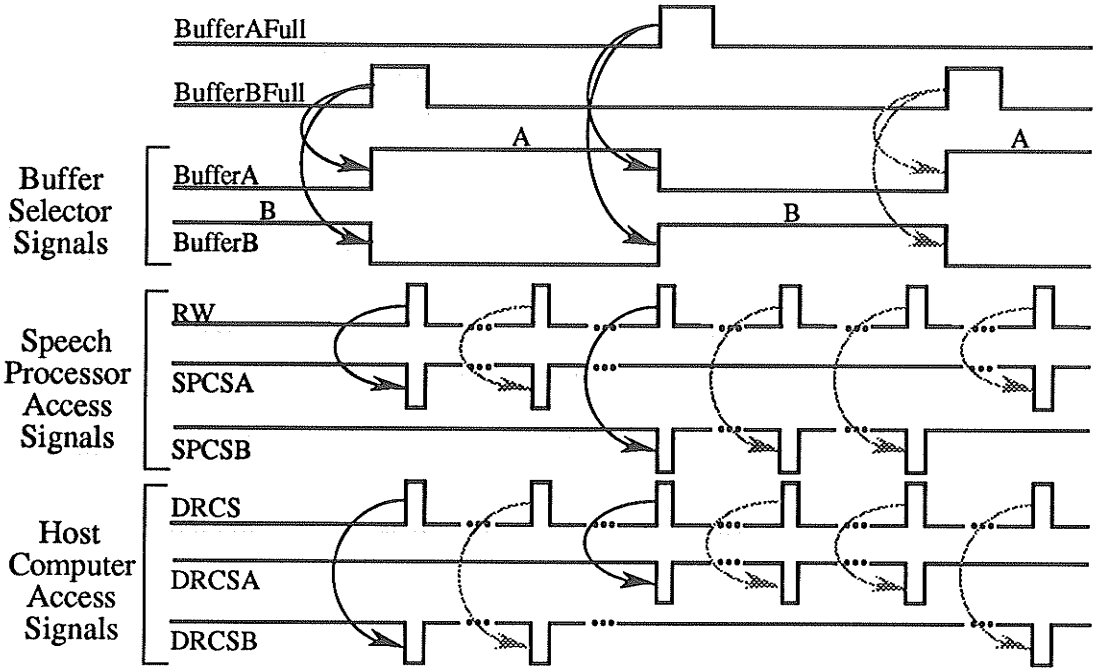


Fig. 5.11 Timing diagram of the swing controller.

In order to determine the device-buffer connection, the swing controller uses buffer full signals. A change in connection, i.e., swing, occurs each time one of the buffers becomes full. 'BufferAFull' and 'BufferBFull' are output signals provided by the FIFO memories, and they indicate when the respective buffer is full of data. Positive edges of these signals cause a swing from one buffer to the other. Causality is indicated in the figure by arrows. 'BufferA' and 'BufferB' are buffer selector signals, and they are used to indicate to which device the respective buffer is connected. A high signal on either Buffer A or Buffer B indicates a connection to the speech processor, whereas, a low signal on either Buffer A or Buffer B indicates a connection to the host computer.

As mentioned previously, the swing controller also derives the buffer chip select signals from RD and WR signals and CS signals associated with speech processor and host computer reads and writes, respectively. The swing controller must always separate the RD and WR signals from the CS signals. This is required to ensure that the two devices do not access the same buffer at the same time.

Figure 5.11 shows how the swing controller derives buffer select signals and achieves separation between speech processor-buffer access and host computer-buffer access. As shown in the figure, the SPCSA and SPCSB signals are used for speech processor access of the buffers. The swing controller derives SPCSA and SPCSB from the logical AND of a buffer selector signal with the Read and Write (RW) signal. Recall from Fig. 5.7 that RW is the logical OR of RD and WR. Thus, RW is active whenever the speech chip is reading from or writing to the buffer. Note that, whenever one of the buffer selector signals is high, the speech processor is accessing data through the assertion of SPCSA or SPCSB.

Similarly, the DRCSA and DRCSB signals are used for host computer access of the buffers. The swing controller derives the DRCSA and DRCSB signals from the logical AND of a logically inverted buffer selector signal with a data register chip select (DRCS) signal. The DRCS signal is generated by the address decoder of the buffer board. Note that, whenever one of the buffer selector signals is low, the host computer is accessing data through the assertion of DRCSA or DRCSB. In this way, the speech processor and the host computer are never accessing the same buffer at the same time.

5.2.3.2 Swinging Buffer Circuit

A circuit that implements the timing diagram described above is shown in Fig. 5.12. As shown in the figure, the swinging buffer consists of two FIFO memories, a self toggling flip flop (FF), and primitive logic gates.

The FIFO used in this design is the WD1510, which is organized as a 9-bit by 128 or 132 word stack. The chip has two bidirectional data ports and may be read from or written into either port. The direction input pin is used to specify the data flow direction.

When it is low, Dir specifies that Port L may be read from and Port R may be written into.

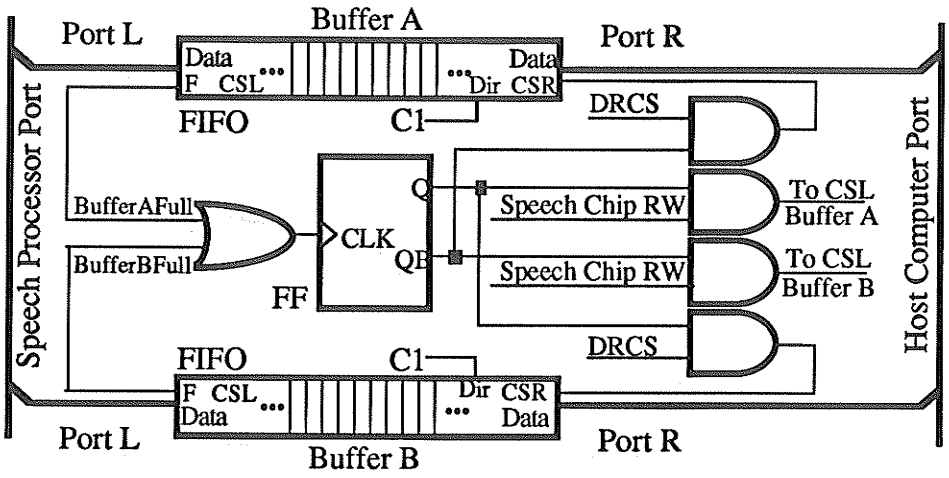


Fig. 5.12 Swinging buffer schematic.

When it is high, Dir specifies that Port L may be written into and Port R may be read from. Reading or writing is performed by setting the appropriate chip select (CSL or CSR) line to logic 0. After the specified hold time (150 nsec) has expired, data may be entered or read at the rising edge of CSL or CSR. Both ports return to high impedance state when CS is returned to logic 1. Reading or writing to the two ports can be done asynchronously. The full output pin is used to indicate when all 128 or 132 words of memory are loaded with data. The empty output pin is used to indicate when there is no data in the buffer [WeDi83].

The purpose of the flip flop is to direct the SPRW and DRCS signals to the chip select input of the appropriate buffer. The FF is triggered by either assertion of BufferAFull or BufferBFull. Note that BufferAFull and BufferBFull cannot be asserted at the same time, since that while the speech chip is filling one buffer, the other buffer is either being read by the host computer or is empty, since the host is much faster than the speech chip.

The operation of the swinging buffer during the record mode is as follows. Because

C1 is logic 1, Port L of Buffer A is selected for write mode, while Port R of Buffer B is selected for read mode. The CSL input signal is active since the SPRW signal is filtered through the logical AND of SPRW with Q. Note that the DRSC signal tending towards Buffer A is blocked since it is paired with QB. On the other hand, the DRSC signal tending towards Buffer B is enabled since it is paired with Q. Therefore, when Q is high, the speech processor may write data to Buffer A, and, if Buffer B contains data (i.e., if Empty is logic 0), then the host computer may read Buffer B. When Buffer A becomes full, the FF triggers, Q goes low and QB goes high, and a swing occurs. That is, the speech processor may now continue writing data to Buffer B, while the host computer may begin reading Buffer A. When the speech processor stops writing data, the host continues reading the buffers, flushing the buffers, until both buffer empty signals are logic 1.

The operation during the playback mode is similar, one difference is the direction logic is reversed by C1, which is logic 0 during playback.

5.2.4 Buffer Schematic

The entire schematic diagram of the speech data buffer board is shown in Fig. 5.13.

5.3 Summary

This chapter presents an alternative paper design of a buffer for intermediating the communication of speech data and control signals between a speech processor and host computer. Rather than using a microprocessor, the speech processor is controlled through a digital circuit consisting of primitive logic gates, address decoder, and external hardware FIFO memory. This design is intended to be implemented using XILINX Logic Cell Array (LCA) technology.

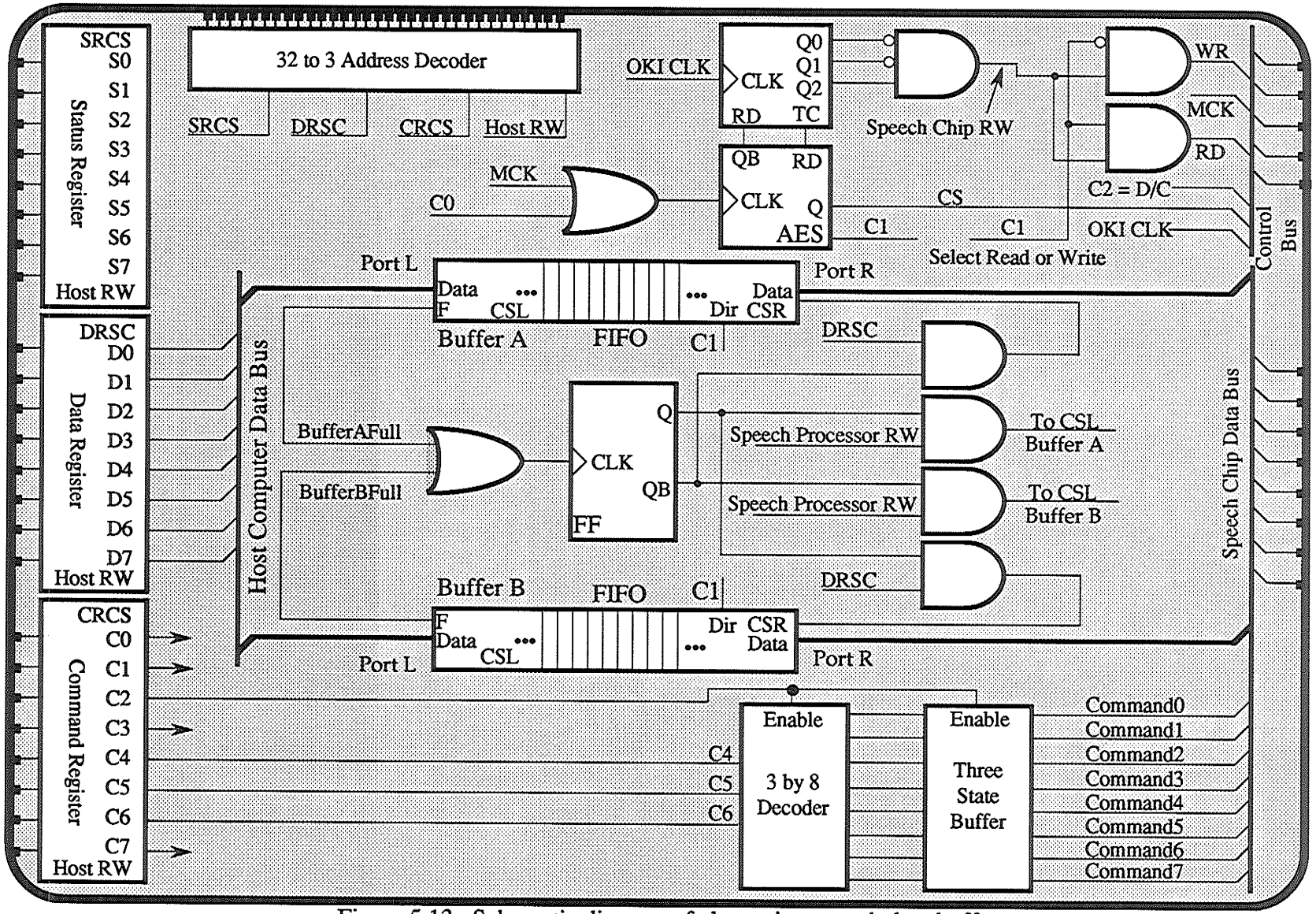


Figure 5.13 Schematic diagram of alternative speech data buffer.

CHAPTER VI
SPEECH SPLICING EXPERIMENTS

Speech splicing is a technique of synthesizing new words or utterances by a process of concatenating the waveforms of component parts. The components form a basis, and the span of their concatenation forms a set of synthesized words. These component parts vary in size, ranging from phonemes to entire syllables (sequences of phonemes). Much like the meaning of words can be changed by changing their phonetic spelling, the sound of new words may be produced by adding, deleting, or substituting component parts. For example, the word 'beet' can be changed to 'feet' by substituting /F/ for /B/. Similarly, we can produce the sound of the new word 'feet' by putting together the sounds /F/ with a combination of /IY/ and /T/ (see Table 2.1 for phonetic transcriptions).

Speech synthesis by waveform concatenation can be utilized in automated vocal shaping systems. Speech splicing techniques can be used to expand the existing library of target sounds. This chapter is designed to show how to use the tools of speech splicing for library expansion in automated vocal shaping systems. Also, preliminary subjective tests are conducted in order to determine the validity of the synthesis methods of interest.

To this end three experiments are performed. The first experiment deals with extracting parts of words of an existing library consisting of naturally spoken words. These extracted parts are processed so that they, alone, may be used for vocal shaping. The second experiment deals with synthesizing new words using an existing library of naturally spoken words as the basis, i.e., the synthesis units that make up the new words are extracted from naturally spoken words. These extracted units include individual phonemes or groups of adjacent phonemes, such as those comprising entire syllables of a word. The third

experiment deals with synthesizing new words using natural phonemes uttered in isolation as the basis, i.e., the synthesis units that make up the new words are naturally spoken phonemes. These experiments use the waveform synthesis tools, as described in Chapter II, Section 2.2.4, namely, copy, cut and paste; amplitude interpolation; and linear predictive extrapolation (LPE).

6.1 Apparatus

The equipment used in these experiments consists mainly of computer and sound production equipment. As such, the apparatus can be classified as either hardware or software.

6.1.1 Hardware Equipment

Figure 6.1 shows a block diagram of the equipment used in the experiments. Note that there are two workstations. The workstation on the IBM computer is the one designed in this thesis, as described in Chapter IV. Not all the capabilities of this system are utilized

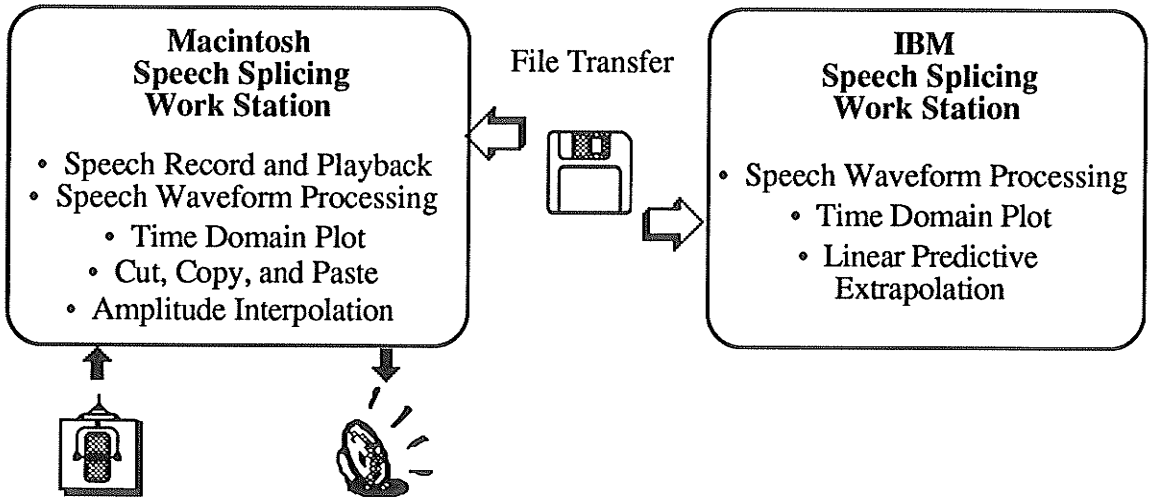


Fig. 6.1 Block diagram of experimental equipment setup.

in these experiments, only the speech waveform processing features are used. This is because the other required functions, such as speech recording and playing and further waveform processing, are easier to perform on the Macintosh computer workstation and, also, recording and playing back PCM data is not possible with the system designed on the IBM, for reasons explained in Chapter IV, Section 4.5. The Macintosh workstation consists of a Macintosh IIsi computer. The IIsi has built in sound I/O, including an electret microphone, speaker, and a minijack output. Rather than the built in speaker, a set of AKG K240 headphones is used for speech playback. The use of headphones decreases the ambient noise (although the ambient noise was low) and facilitates a better environment for subjective testing.

6.1.2 Software Tools

The software used on the IIsi workstation consists of the Macintosh Sound Manager, Audio stack of HyperCard 2.0, SoundEdit™ 2.0, Apple File Exchange, and ResEdit 2.1. The software responsible for executing the linear predictive extrapolation method is located on the IBM workstation, and the source code for this software can be found in Appendix A2.11.

6.2 Method

The IIsi is used to record speech and store its waveform on hard disk. The Sound Manager or the Audio Stack of HyperCard 2.0 are used for recording speech. A limited recording time (sufficient for recording medium length sentences) is allowed, and speech data can be saved as a resource file in either the System or in one of HyperCard's stacks, depending on the software used. Speech is recorded at a sampling frequency of 22 kHz and coded in PCM format (no compression). Once speech data is saved on hard disk, then

the data file may be opened by the application SoundEdit. A wide variety of speech processing tools and features are made available by SoundEdit, such as the copy, cut, and paste and amplitude interpolation synthesis methods and a visual association between the waveform and sound of speech. SoundEdit also allows saving files in various formats, including resource, SoundEdit, and, particularly, AIFF, the purpose of which is discussed in Section 6.2.1.

6.2.1 Macintosh to/from IBM Speech Data File Transfer

Apple File Exchange and ResEdit are used to facilitate file transfer between Macintosh files to and from IBM files. An IBM formatted floppy disk (3.5", 1.44 MByte) is used for the transfer. Note that the disk is required to be of 1.44 MByte capacity because the Macintosh IIsi uses a 1.44 MByte SuperDrive. In order to prepare for disk transfer, the Macintosh file is first saved in Audio Interchange File Format 1.33 (AIFF) by SoundEdit. The use of AIFF is important because the LPE software of the IBM workstation changes the size of the speech data file, and this information must be included in the header information of the AIFF formatted file in order that SoundEdit be able to read the resulting modified file. Having saved the file in AIFF format, the file is then transferred to an IBM formatted disk using the *Default Translation* of Apple File Exchange.

Because the LPE software changes the size of the file, associated software, also written on the IBM workstation, is used to update the size information in the file header, which is shown in Fig. 6.2. The header information is updated in three places, *ckSizeFORM*, *numSampleFrames*, and *ckSizeSSND*, and these locations are changed according to equations as shown at the bottom of the figure. Prediction Length is the amount of bytes added to the file by the LPE program.

After running the LPE program and updating the header information to reflect the

change in size of the file, the file is transferred back to the Macintosh IIsi workstation. Once again, Apple File Exchange is used for the transfer. However, after the transfer is

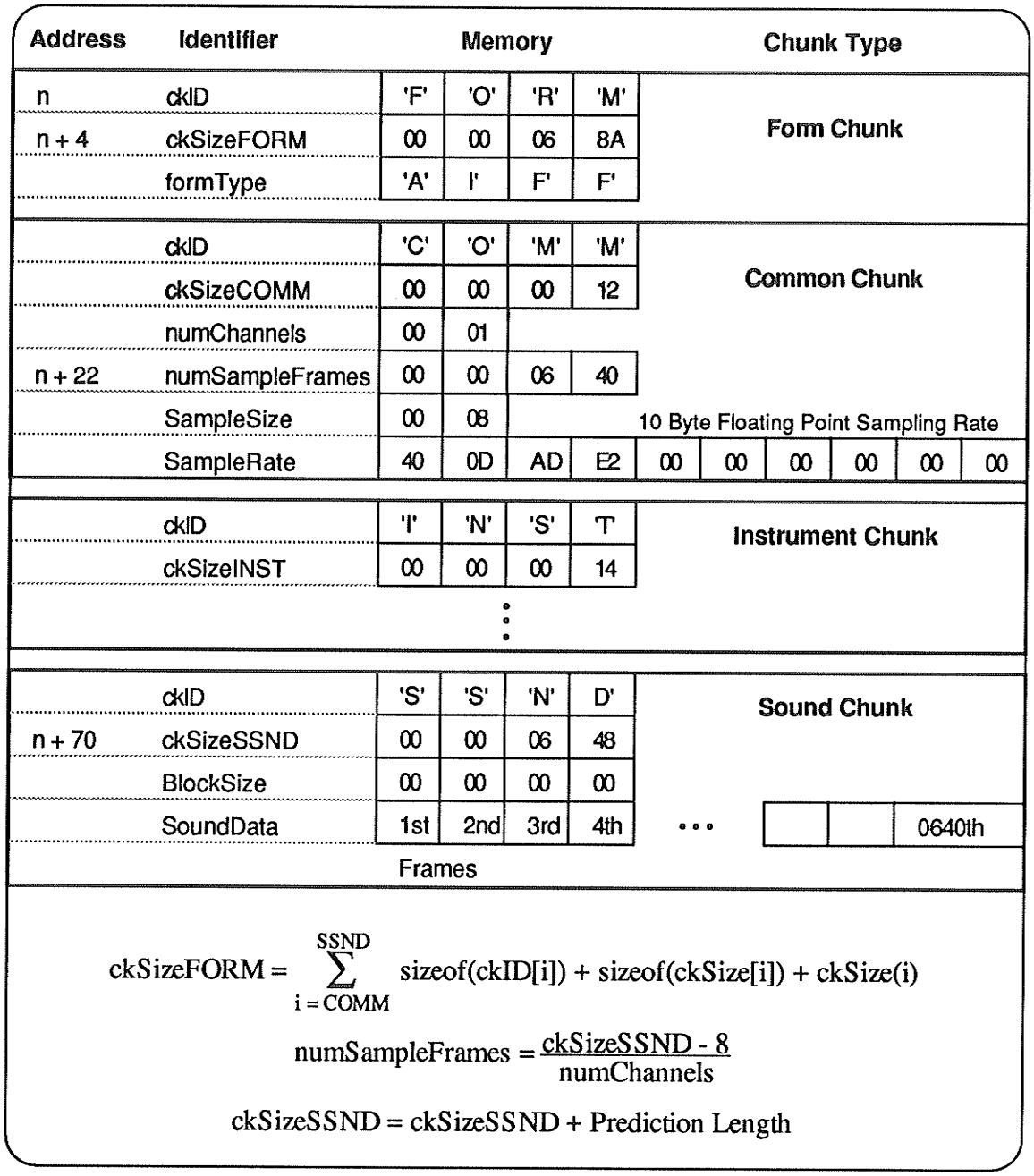


Fig. 6.2 AIFF header format.

complete, ResEdit is used in order to change the type and creator information of the file. The default translation of Apple File Exchange creates a file that is of type binary and

created by MDOS. In order that SoundEdit be able to reopen the LPE modified file, the type must be changed to AIFF and the creator must be changed to SFX!. ResEdit can be used to do this. The ResEdit window in which this is done is shown in Fig. 6.3.

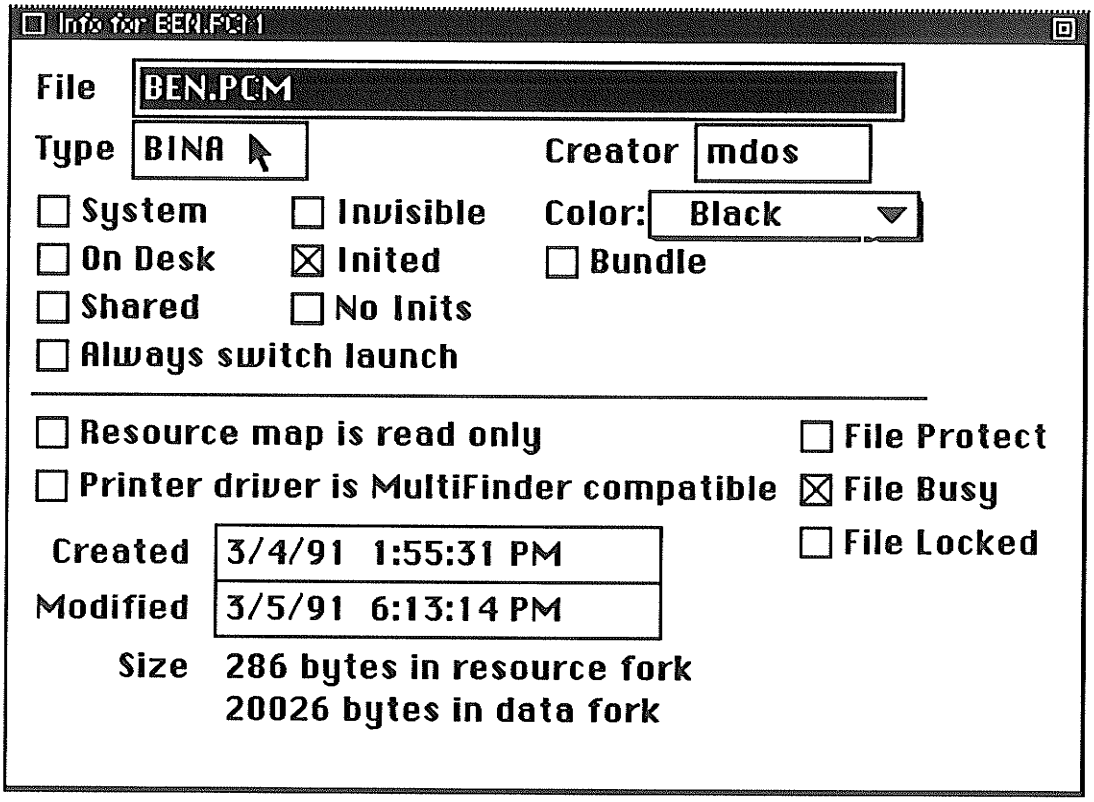


Fig. 6.3 ResEdit window for changing file flag information. In order that SoundEdit be able to open the modified file, the type BINA must be changed to AIFF and the creator mdos must be changed to SFX!.

6.2.2 Verification of Linear Predictive Extrapolation (LPE) Software

A method for verifying the LPE software is as shown in Fig. 6.4. In addition to predicting a number of future samples, the algorithm also 'predicts' the sample frame from which the predictions are made. The prediction of the sample frame is compared to the real sample frame in two respects, visually and in the mean square error (MSE) sense. The visual comparison provides a rough subjective confirmation, and the MSE provides an objective verification that the software is working.

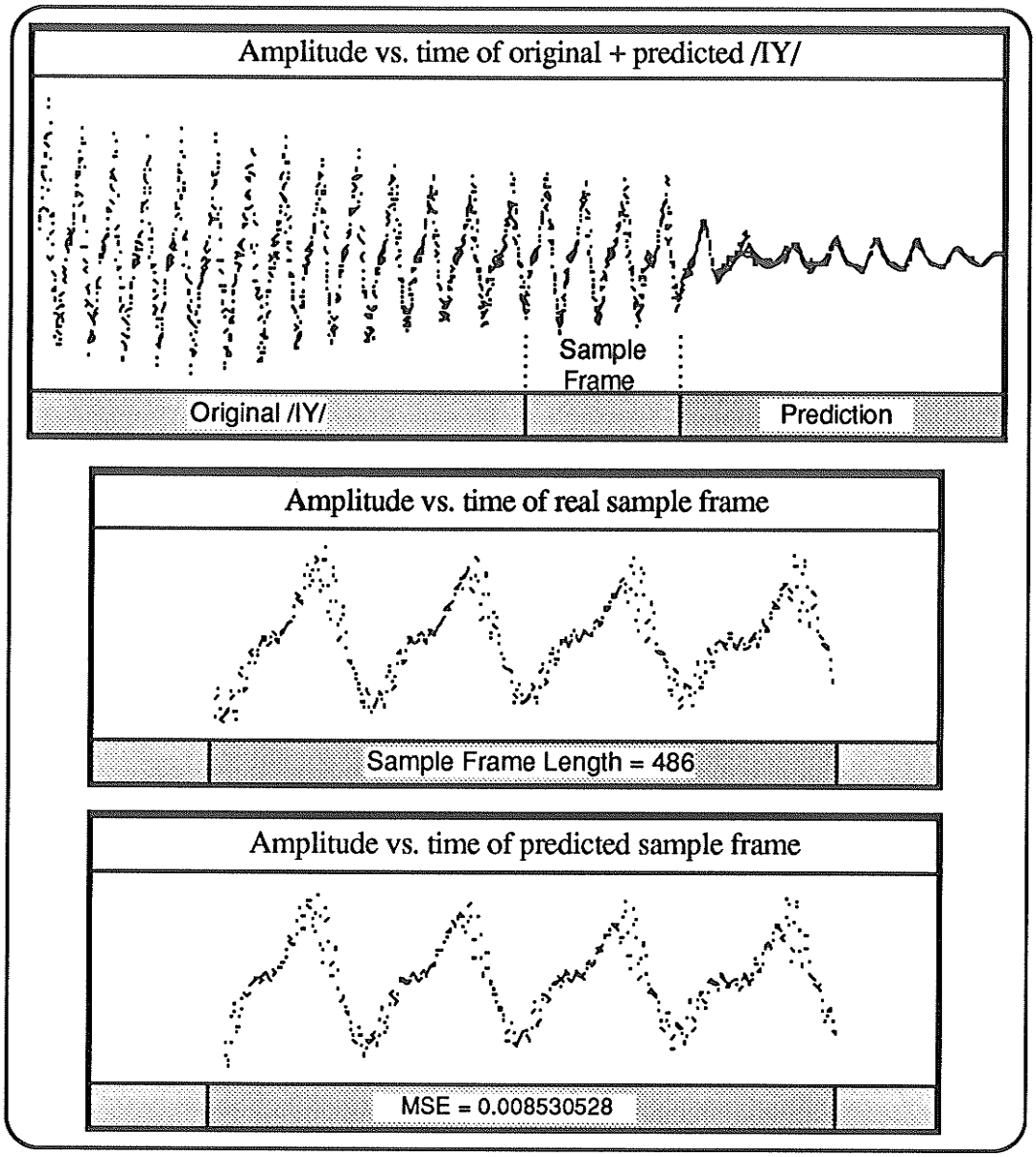


Fig. 6.4 LPE software verification. Mean square error (MSE) of .8% is an objective verification that the LPE software is working.

6.2.3 Expansion by Extraction

The purpose of the first experiment is to show how to use speech splicing techniques in order to expand an existing library of words spoken by a therapist. The objective is to extract phonemes or syllables in order that they, themselves, may be used for vocal shaping. The idea is that if a student is to learn to vocalize an entire word, it would be

easier to proceed in steps by individually shaping the component parts of the word. Once the component parts are learned, then they may be put together more easily in order to form the word.

For this experiment, it is assumed that a library of words exists, and that this library contains the words, feet, fit, Ben, and well (randomly chosen). From these words the following phonemes and syllables are extracted: /F/, /B/, /W/, /T/, 'fee', 'eet', 'it', 'en', and 'ell'. From these component parts all of the existing words may be formed, in addition to other words, such as, bit, bell, beet, fell, and wheat.

Figure 6.5 shows an example of the procedure involved in extracting the phonemes /F/, /IY/, and /T/ and the syllable 'eet' from the word 'feet'. SoundEdit is used to open and display the time domain plot of the previously recorded file containing the word 'feet'. The sound of each individual phoneme and syllable is associated with its waveform by selecting a portion of the waveform, i.e., dragging over the waveform using the mouse. The selection is then played. This association can be done by trial and error, but more educated guesses can be made by realizing the properties of the component parts. The phoneme /F/ is called an unvoiced fricative mainly because it sounds like noise, as it is produced by forcing air through the spacings of one's teeth. This is in contrast to the vowel phoneme /IY/, which is a periodic sound, as it is produced by vibrating one's vocal cords. Indeed, as shown in Fig. 6.5, this contrast is also exhibited in the waveform. Furthermore, because the phoneme /T/ requires that all sound production stop before it can be produced, we expect that the waveform preceding /T/ to be relatively flat and have constant amplitude near zero volts. Note in the figure that the periodic sound of /IY/ tapers off to zero volts and stays there for some time before the waveform for /T/ is produced.

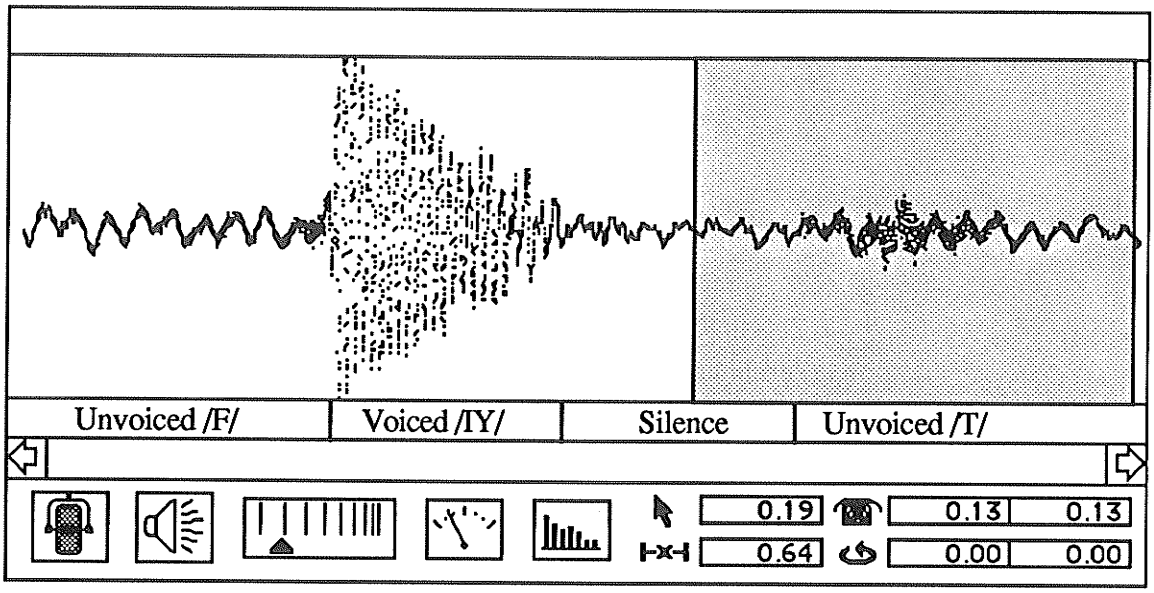


Fig. 6.5 Waveform of 'feet'. This figure is a snapshot of a window produced by SoundEdit.

Once the waveforms of the component parts have been identified, then they may be played back for vocal shaping purposes. However, more processing may be needed in order to smooth the boundary conditions so that there is no coarse beginning or ending sound. This processing may merely involve selecting an appropriate segment. As shown in Fig. 6.5, the selection of /T/ includes silence on both sides of the actual waveform. The waveform of this selection is referred to as the diphone of /T/, since the boundary conditions on either side of /T/ are characterized by steady state regions, i.e., relative silence. Extracting diphones in this way can be done with all unvoiced stops (/T/, /P/, and /K/) uttered in the context of words.

More involved processing may be needed for other types of boundary conditions. For example, extracting /IY/ from Fig. 6.5 may require processing at the beginning of the sound in order to eliminate the abrupt beginning. This can be done in at least two ways.

The first method involves using amplitude interpolation. More specifically,

SoundEdit offers an amplitude envelope function which can be used to scale down the envelope at the beginning of the /IY/ sound, as shown in Fig. 6.6.

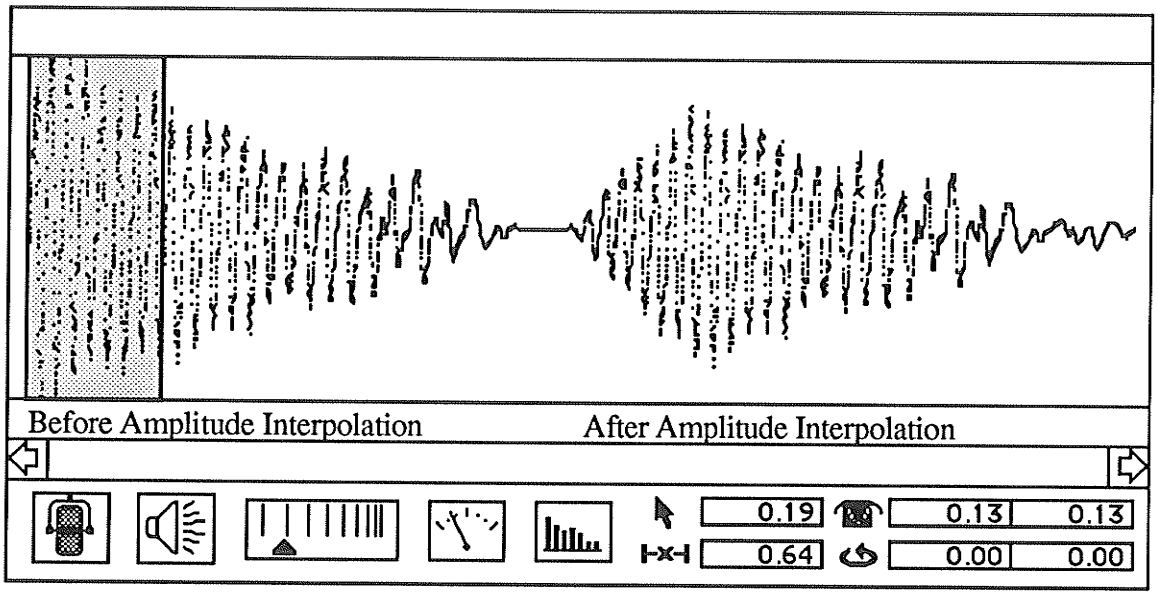


Fig. 6.6 Amplitude Interpolation of /IY/.

The second method involves using LPE to postdict a number of samples. As mentioned above, LPE synthesis software is located on the IBM workstation, and, therefore, the SoundEdit file of 'feet' must be saved in AIFF format, transferred to the IBM workstation using Apple File Exchange, operated on by LPE software on the IBM workstation, updated to reflect its file size changes, transferred back to the Macintosh IIsi workstation, updated to reflect its new file flag information, and, finally, opened, once again, by SoundEdit. The time required for the entire process depends largely on the number of samples to be predicted or postdicted by the LPE software, e.g., for a 500 point postdiction, this requires about 30 min. Fig. 6.7 shows the result of postdicting 500 samples of the phoneme /IY/ of Fig. 6.6.

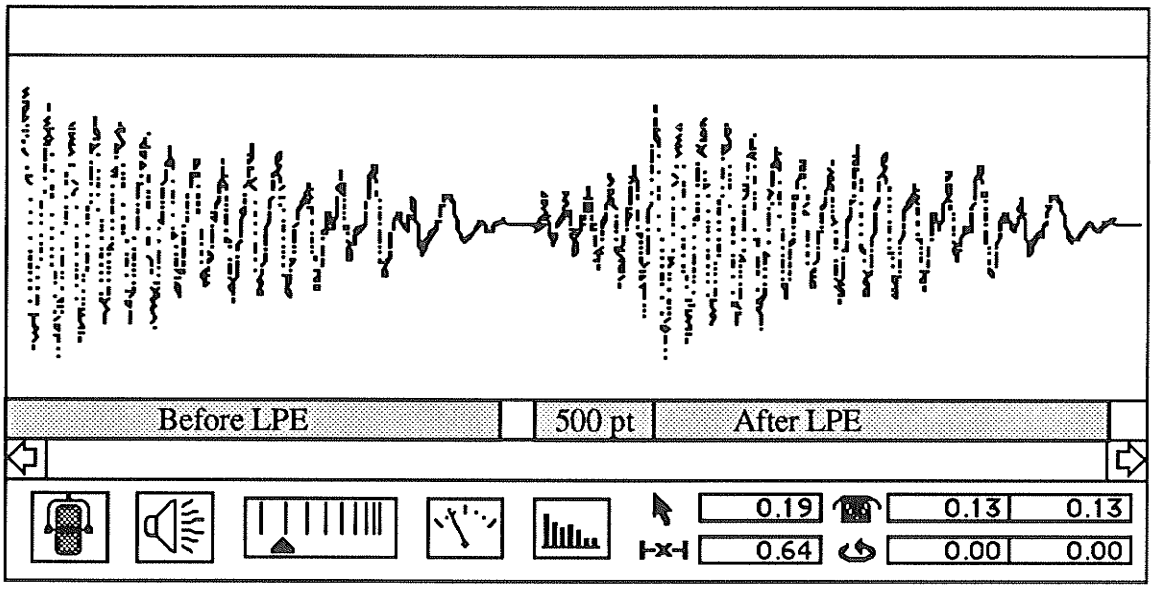


Fig. 6.7 Postdiction of phoneme /IY/.

Amplitude interpolation and linear predictive extrapolation, as used in the above two examples, maintain continuity of the waveform. This is an objective validation of the methods. Furthermore, they improve the overall sound of the extracted components, while not altering intelligibility. That the above synthesis methods are subjectively valid is examined in the following two experiments.

6.2.4 Expansion by Sub-word Splicing

The purpose of the second experiment was to conduct subjective tests on new words formed by splicing together phonemes and/or syllables extracted from a library of existing words. The tests consisted of quality assessment and preference. Ten people from the University of Manitoba, including three electrical technicians and seven students were chosen to participate. Two of the student participants were currently doing speech related research, four were from the faculty of Electrical and Computer Engineering, and one was from the faculty of Business and Administration. Two of the participants were women.

The existing library consisted of the words, 'beet', 'bell', 'fit', and 'when', which were vocalized and recorded by the author. The synthesis units were phonemes and/or sequences of phonemes extracted from these words, and they were put together using the above mentioned synthesis techniques, namely, copy, cut, and paste; amplitude interpolation; and linear predictive extrapolation. The new words formed along with their synthesis units were as follows:

/F/ + 'eet' = 'feet',

/B/ + 'en' = 'Ben',

/W/ + 'ell' = 'well', and

/F/ + 'ell' = 'fell',

/W/ + 'eet' = 'wheat',

/B/ + 'it' = 'bit'.

The participants were seated nearby the Macintosh IIsi workstation, and they were supplied with headphones and response sheets, as shown in Fig. C1, Appendix C. There were two tests, quality assessment and preference.

In the quality assessment test, the participants listened to two sets of words, one set was the spliced words, as mentioned above, and the other set consisted of the same words, the only difference was that these words were naturally formed and also recorded by the author. The purpose of this second set was to obtain a normalized scale, so that the responses made by the participants would be judged relative to what they thought was natural. In both of these tests, the participants were asked to indicate the quality by placing a mark in the adjoining rectangle, as shown in Fig. C1a and Fig. C1b.

In the preference test, three versions of four words, 'feet', 'fell', 'wheat', and 'well' were used. The first version of each word was spliced together using only the copy, cut, and paste method of synthesis. The second version of each word was a boundary modification of the first version using amplitude interpolation.

The third version of each word was a boundary modifications of the first version using the LPE binding segment method, as described in Chapter II, Section 2.2.4.3. For the words 'feet' and 'fell', the binding segment was produced in two steps as follows: 500 points (23 ms) were postdicted from the beginning sounds of the extracted components 'eet' and 'ell'. The postdicted frame was then averaged point by point with 500 samples of the ending of the phoneme /F/. The resulting segments were inserted between the phoneme /F/ and the components 'eet' and 'ell', thus forming the words 'feet' and 'fell'.

For the words 'wheat' and 'well', the binding segment was produced in three steps as follows: 500 points (23 ms) were predicted from the ending sound of the semivowel phoneme /W/. Also, 500 points (23 ms) were postdicted from the beginning sounds of the extracted components 'eet' and 'ell'. The postdicted frame was then averaged point by point with the predicted frame. The resulting segments were inserted between the phoneme /W/ and the components 'eet' and 'ell', thus forming the words 'wheat' and 'well'.

For example, Fig. 6.8a shows the waveforms after prediction and postdiction of /W/ and 'eet', respectively. Figure 6.8b shows the resulting waveform after averaging the predicted and postdicted frames. Note that after averaging, the waveform of the binding segment was amplified in order to bring the amplitude to a level comparable to the waveforms on either side of it.

For the above three versions of the four words, 'feet', 'fell', 'wheat', and 'well', the participants were asked to indicate their preference on response sheets, as shown in Fig. C1c. The purpose of the preference test was to determine whether amplitude interpolation or LPE improved the quality of the splice.

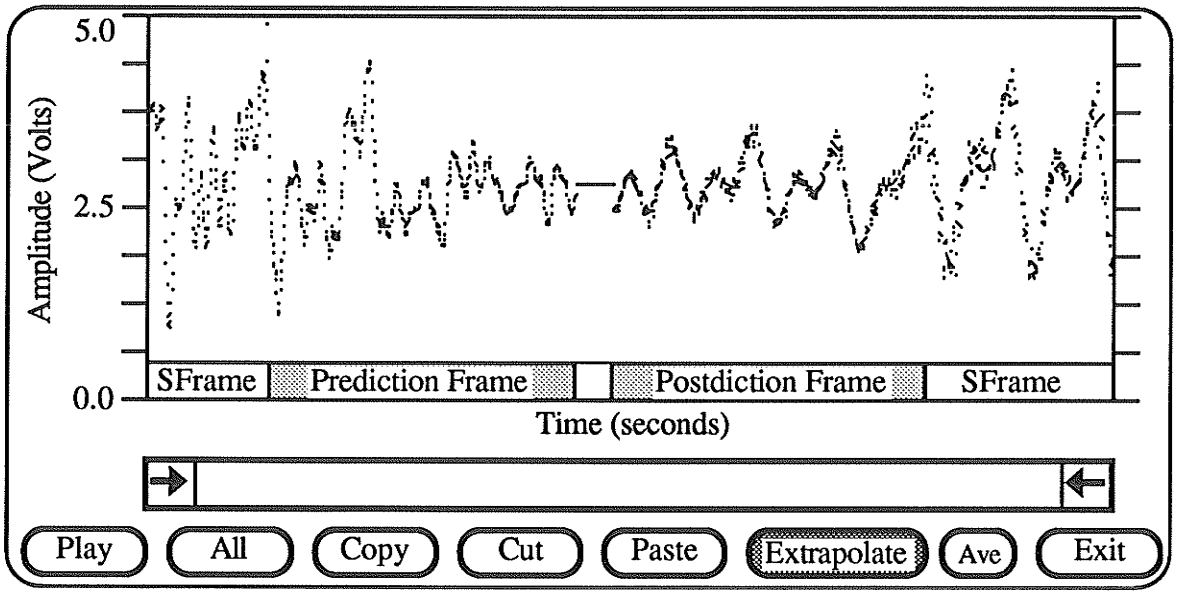


Fig. 6.8a Prediction of phoneme /W/ (left) and postdiction of 'eet' (right).

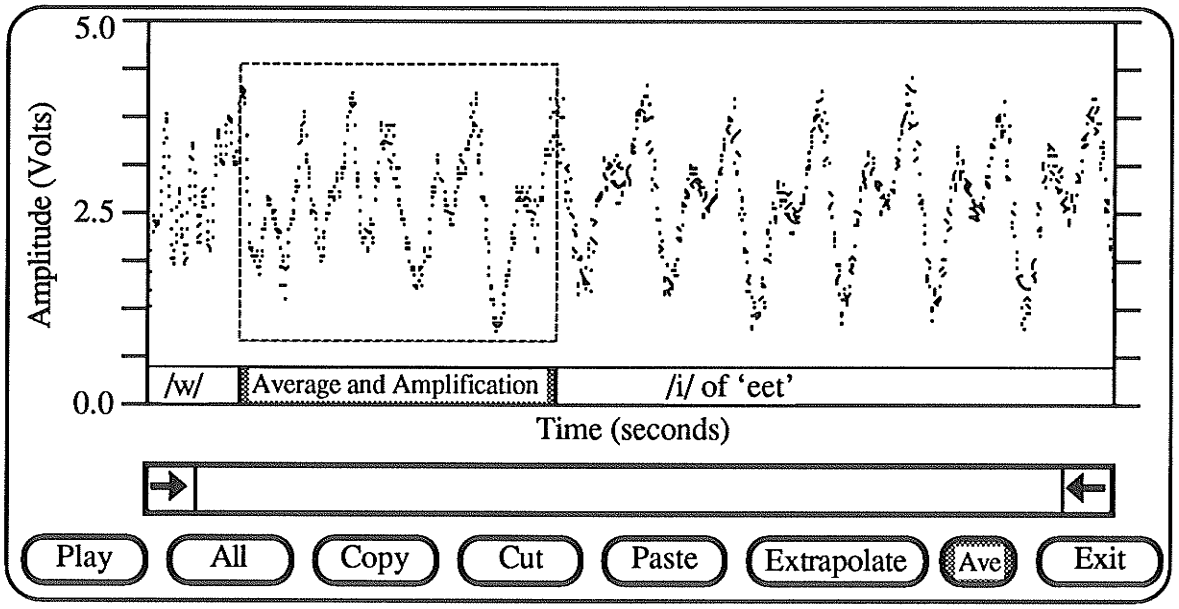


Fig. 6.8b Averaging of prediction of /W/ and postdiction of 'eet'.

6.2.5 Expansion by Phoneme Splicing

The purpose of the third experiment was to conduct subjective tests on new words formed by splicing together isolated phonemes from an existing library. The tests consisted of quality assessment, similarity, and preference. The same ten people participated.

The existing library consisted of the phonemes, /B/, /F/, /T/, /S/, /W/, /K/, /IY/, /UW/, /EA/, /IH/, and /EH/. These phonemes were recorded in isolation by the author. These synthesis units were put together using the above mentioned synthesis techniques, namely, copy, cut, and paste; amplitude interpolation; and linear predictive extrapolation. The new words formed along with their synthesis units were as follows:

/B/ + /EA/ + /N/ = 'Ben',

/B/ + /UW/ + /T/ = 'boot',

/F/ + /IY/ + /T/ = 'feet',

/S/ + /IH/ + /T/ = 'sit',

/F/ + /IH/ + /T/ = 'fit'.

/B/ + /EH/ + /T/ = 'bet',

/K/ + /AE/ + /T/ = 'cat',

/W/ + /EH/ + /T/ = 'wet'.

/B/ + /IY/ + /T/ = 'beet', and

The participants were seated nearby the Macintosh IIsi workstation, and they were supplied with headphones and response sheets, as shown in Fig. C2, Appendix C. There were three tests, the first of which was quality assessment, followed by preference, and finally, similarity.

In the quality assessment test, the participants listened to the set of spliced words, as mentioned above. From a given list of words, they were asked to indicate the quality on the sheet of Fig. C2b.

In the preference test, two versions of the four words, 'Ben', 'boot', 'cat', and 'wet' were used. The first version of each word was spliced together using the copy, cut, and paste and amplitude interpolation methods of synthesis. The second version of each word was a boundary modifications of the first version using the LPE binding segment method, as implemented in the previous experiment.

For the words 'Ben', 'boot', 'cat' and 'wet', two binding segments were formed and placed at the boundary between the first and second and second and third phonemes. The type of binding segment formed depended on the type of phonemes to be joined with the center vowel phoneme. If the boundary properties of the abutting phonemes were similar, then a three step method was used.

For the synthesis involving the joining of /B/, /N/, /K/ or /T/ with a vowel phoneme, a two step procedure was used as follows: 500 points (23 ms) were postdicted and predicted from the phonemes /E/, /OO/, and /AE/ (e.g., vowel sounds in 'B*e*n', b*oo*t', and 'c*a*t', respectively). The postdicted and predicted frames were then averaged point by point with 500 samples of either the ending or beginning sounds of /B/, /N/, /K/ or /T/, correspondingly.

For the synthesis involving the joining of the semivowel /W/ with the vowel phoneme /E/, a three step procedure was used as follows: 500 points (23 ms) were predicted from the ending sound of /W/. Also, 500 points (23 ms) were postdicted from /E/. The postdicted frame was then averaged point by point with the predicted frame. The resulting segment was inserted between the phonemes /W/ and /E/, thus forming the word 'wet'.

For the above two versions of the four words, 'Ben', 'boot', 'cat', and 'wet', the participants were asked to indicate their preference on response sheets, as shown in Fig. C1c. The purpose of this preference test was to determine whether LPE improved the quality of the splice.

In the similarity test, for each of the vowel phonemes, /IY/, /U/, /AE/, /E/, /I/, /OO/, and /OW, the participants were asked to judge the similarity between a 25 ms sample frame of a phoneme followed by its 25 ms prediction. They were also asked to judge the

similarity between the same 25 ms sample frame and its 25 ms postdiction. Participants were asked to indicate the similarity on sheets as shown in Fig. C1a. The purpose of this test was to determine the subjective validity of the linear predictive extrapolation method.

6.3 Presentation and Analysis of Results

After the tests were completed, all response sheets were gathered for observation and analysis. For the quality assessment and the similarity tests, the response rectangles were quantized into 8 uniform levels, one being the lowest and eight being the highest. The quantized values were tabulated and averaged over the ten participants. The average values were then plotted in bar graphs.

6.3.1 Extracted Sub-word Splicing

Figure 6.9 shows the assessment of words formed by concatenating phonemes and/or sequences of phonemes extracted from an existing library of natural words. The copy, cut, and paste synthesis method was used. As can be seen in the figure, the spliced words show a high degree of naturalness, except for, perhaps, the words 'well' and 'wheat'. The high degree of naturalness is attributed to the type of units employed in the synthesis. For each of the words, 'bit', 'Ben', 'fell' and 'feet', two synthesis units were used, and they consisted of a consonant phoneme spliced together with a sequence of two phonemes, the first of which was a vowel phoneme. As such, there existed a great degree of discontinuity at the boundary of the splice, because of the very different nature of the abutting phonemes. Recall the natural recording of 'feet' in Fig. 6.5, where the discontinuous boundary also exists between the phonemes /F/ and /IY/.

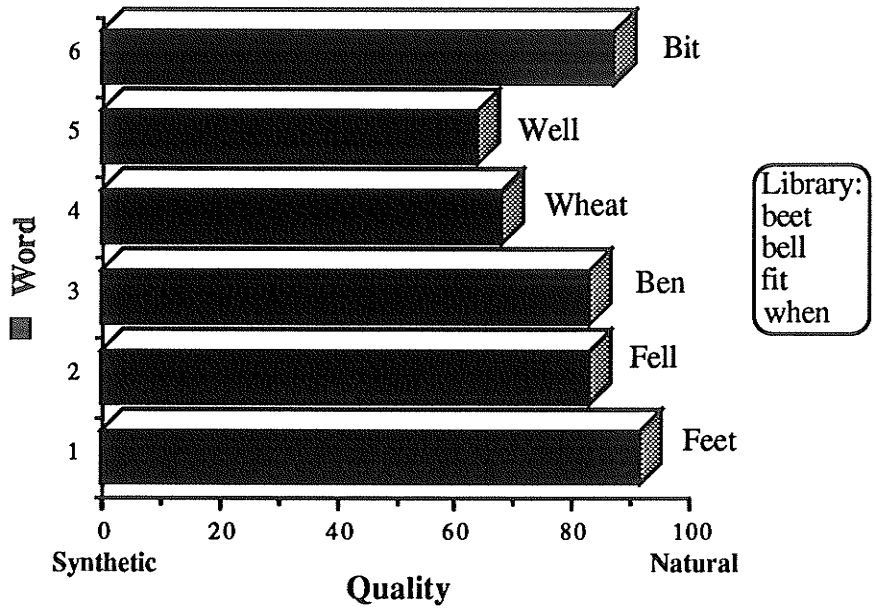


Fig. 6.9 Subjective response to word synthesis by extracted phoneme concatenation.

While the words 'well' and 'wheat' can be regarded as showing a certain degree of naturalness, they are singled out, because their naturalness is about 15% lower than the others. This lower quality is also attributed to the types of units used in the splice. In particular, the semivowel phoneme /W/ was put together with a sequence of two phonemes, the first of which was a vowel phoneme. Because the boundary properties of the abutting phonemes were similar (e.g., compare the waveforms of /W/ with /IY/ in Fig. 6.8b), achieving a natural sound was more difficult. This problem may be the manifestation of coarticulation. For example, in the natural vocalization of the word 'wheat', the ending of the phoneme /W/ is influenced by and influences the following beginning of the phoneme /IY/. Furthermore, that /W/ was taken from the naturally recorded word 'when', complicates the problem further for the spliced word 'wheat'. This is because /W/ had been influenced by /E/. As a result, if this type of splice is to sound natural, the influence of /E/ on /W/ will have to be removed and the interaction between /W/ and /IY/ will have to be introduced. Or, alternatively, a /W/ uttered in isolation can be used. This reduces the problem to introducing the interaction.

6.3.2 Isolated Phoneme Splicing

Figure 6.10 shows the assessment of words formed by concatenating isolated phonemes using the copy, cut, and paste method. As can be seen in the figure, the participants indicated that the words sounded more synthetic than natural. In addition to the coarticulation problem as discussed above, this is attributed to the lack of correct timing, stress, pitch adjustment, and intonation that is characteristic of phonemes uttered in isolation. For example, the phoneme /I/ uttered in isolation is typically lower in pitch and lacks the stress required by the vowel in 'fit'. Splicing phonemes uttered in isolation is more difficult than splicing extracted units, as described above, simply because the units are smaller and more processing, that the vocal tract would otherwise have done, must now be considered by the synthesizer.

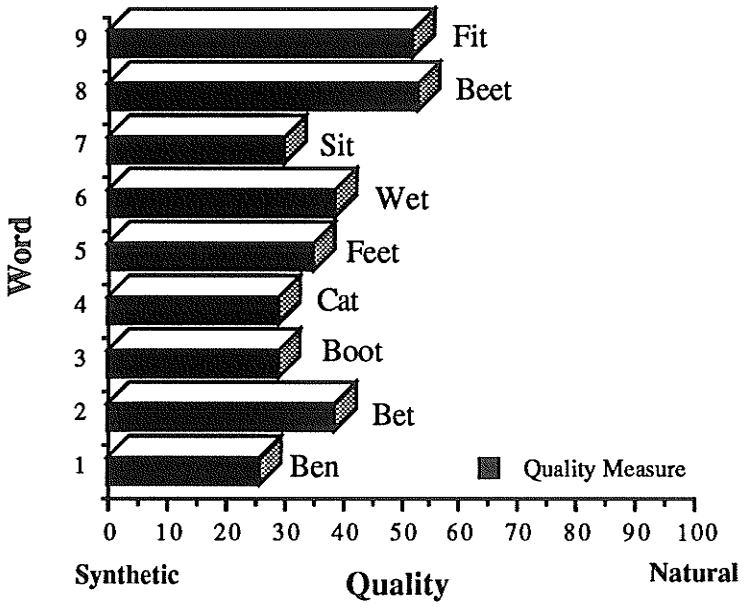


Fig. 6.10 Subjective response to word synthesis by isolated phoneme concatenation.

6.3.3 Similarity

Figure 6.11 shows results of the similarity test. These results show the degree of

similarity between 25 ms of an original vowel phoneme and 25 ms of the predicted and postdicted versions. As can be seen, the participants indicated both predictions and postdictions sounded fairly similar to the originals. It can be argued that the lack of exact similarity may be because the time, 25 ms, is too short for making a judgement and that the subjects were forced to make a random decision. If so, this may be a source of experimental error. Also, it can also be argued that the lack of exact similarity is due to the assumptions and estimations made by the linear predictive sub-model of speech. In particular, the all-pole model ignores nasals. Furthermore, the covariance method has been shown to model periodic speech sounds better than the autocorrelation method. These arguments are supported by the fact that linear predictive coding, when used as a compression technique, achieves relatively low quality, as shown in Fig. 2.3. However, these results, in addition to the results shown in Fig. 6.4, suggest at least that the LPE software is working.

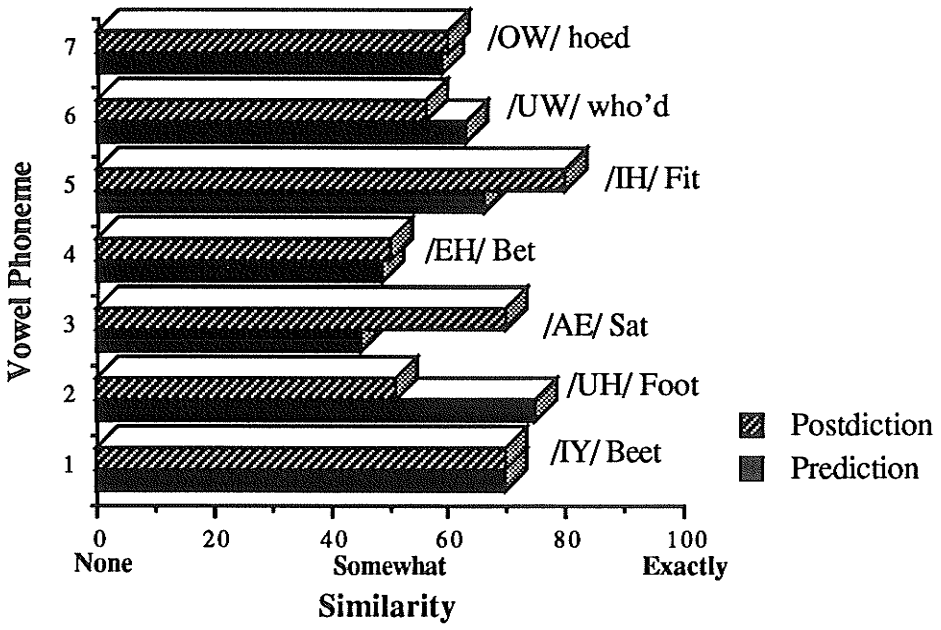


Fig. 6.11 Subjective response to 25 ms phoneme prediction and postdiction.

6.3.4 Preference

Table 6.1 shows the results of the preference test for words formed by

concatenating extracted subunits of existing words, as described in Section 6.3.1. The purpose of this test was to determine the effectiveness of the boundary modification methods, namely, amplitude interpolation (version 2) and linear predictive extrapolation (version 3). Version 1 is formed by the straight forward copy, cut, and paste method. As can be seen in the table, version 2 (amplitude interpolation) was the preferred choice for every word. This suggests an incremental improvement over the corresponding words and their quality ratings of Fig. 6.9, because those observations were made with version 1.

Table 6.1 Preference test for words formed from extracted word subunits.

Word	Preference		
	Version 1	Version 2	Version 3
Feet	3	5	2
Fell	2	7	1
Wheat	2	7	1
Well	4	5	1

While version 2 showed the greatest percentage of preference for words formed from extracted sub-word units, the results shown in Table 6.2 suggest that LPE is better than amplitude interpolation for boundary modifications of words formed by concatenating isolated phonemes. In this table version A is formed by using copy, cut, and paste and amplitude interpolation methods of synthesis. Version B is an LPE modification of version A.

These results appear to suggest that, for the words that were tested, LPE is more effective in creating a natural sounding binding segment when the boundary problem is poor initially.

Table 6.2 Preference test for words formed from isolated phonemes.

Word	Preference	
	Version A	Version B
Ben	3	7
Boot	3	7
Cat	4	6
Wet	6	4

6.4 Summary

This chapter describes the apparatus and method, as well as presents and analyzes the results of three preliminary speech splicing experiments. Two speech splicing workstations are used. The Macintosh IIsi workstation is used for main processing and splicing, including speech recording and playback and implementation of the copy, cut, and paste and amplitude interpolation methods of synthesis. The IBM workstation, the speech splicing system designed in this thesis, is used for performing the linear predictive extrapolation synthesis method. File transfer between the Macintosh and the IBM is explained.

The first experiment shows how to expand an existing library for vocal shaping. Phonemes and/or sequences of phonemes (comprising syllables) are extracted in order that they, themselves, may be used for vocal shaping. It is shown how the three methods of synthesis, namely, copy, cut, and paste; amplitude interpolation; and linear predictive extrapolation, may be used for this purpose.

The second experiment formed new words by concatenating the waveforms of sub-word units extracted from existing words. Subjective tests were conducted, and the

preliminary results indicate up to 80% natural quality. Further tests done on the same new words with boundary modifications seem to indicate an incremental improvement when amplitude interpolation is used, at least for the synthesized words of interest.

The third experiment formed new words by concatenating the waveforms of isolated phonemes. Results of subjective testing indicate poor quality, and this may be attributed to coarticulation, the lack of correct timing, stress, pitch adjustment, and intonation.

The experiments performed in this chapter are preliminary tests and no significant exptrapolation of the results is intended. However, for the specific words tested, the preliminary results appear reasonable and may be good indicators for other similar words. More formal testing procedures of speech synthesis can be found in [Klim87] and [Weir82].

CHAPTER VII
CONCLUSIONS AND RECOMMENDATIONS

(A summary of what went on before.

A determination or judgement arrived at by reasoning and investigating)

The work described in this thesis was motivated by the need for a computer aided speech splicing system for vocal shaping. A study of the **psychological, theoretical**, and **technical** aspects of the problem has led to the **development** of a **PC based system** for **speech processing** (i.e., recording, compressing, editing, splicing, synthesizing, and playing) and a **methodology** of **splicing speech** for **vocal shaping.**

A study of the psychological aspect has revealed *potential* advantages of employing speech synthesis tools in automated vocal shaping systems. Speech synthesis may provide experimental **continuity** by target **expansibility**. The introduction of a new teacher, which includes a different pronunciation, coarticulation, and intonation, introduces experimental variables, whose effect may not be fully understood. A desirable synthesis tool is one that is capable of **continuing** the experiment in the absence of the original therapist and **expanding** the target word library while maintaining similarly sounding features.

However, the familiarity provided by similarly sounding features in new words may be an important psychological influence in itself. New words or utterances may be learned more easily if they bear similar and individualistic properties of previously learned targets, which include targets previously spoken by a therapist and, perhaps, previously learned utterances of the student. Furthermore, words may be learned more easily by individually shaping the component parts. But the component parts must be exemplified as they are pronounced in the context of the word. And they must be pronounced identically many times in succession. A desirable synthesis tool is one that is capable of **extracting** component parts consistently and identically as they sound in the context of a previously recorded word and **concatenating** a sequence of component parts in order to form new words or utterances.

A study of the theoretical aspect has determined optimal speech synthesis techniques, tools, and units for vocal shaping. Because digital **waveform techniques** provide the best quality in the speech coding spectrum, and because waveform recordings preserve all of the individualistic properties of a person's speech, such as pitch, stress, intonation, and inflection, this supports their use in a vocal shaping environment. Moreover, the storing of a direct waveform representation of speech facilitates the extraction of component parts, along with contextual information, which is embedded in the waveform.

The **optimal synthesis unit** depends on the application. For small vocabulary systems, such as vocal shaping, the synthesis units are component parts extracted from digitally recorded words or utterances. These component parts include phonemes, diphones, and/or sequences thereof, such as those comprising entire syllables. New words are formed by concatenating the waveforms of these units. For large vocabulary systems, which also include vocal shaping, the synthesis units are isolated phonemes. Forming new words by concatenating the waveforms of isolated phonemes is difficult because of coarticulation and the lack of correct timing, stress, pitch adjustment, and intonation that is characteristic of phonemes uttered in isolation.

Three **synthesis tools** supportive of waveform concatenation are copy, cut, and paste; amplitude interpolation; and linear predictive extrapolation. Copy, cut, and paste facilitates electronic editing of digitally recorded speech, while amplitude interpolation and linear predictive extrapolation are used to modify the boundary properties between synthesis units.

A study of the technical aspect has led to the development of a speech processing system built around a PC AT. The system consists of a dedicated ADPCM speech processor chip (MPU interface version), dual-pointer FIFO buffer, and PC AT host

computer. The use of ADPCM for encoding and decoding speech data is supported by the fact that ADPCM is a waveform technique and that it reduces the cost of transmission by a ratio of 2:1, while maintaining good toll quality. The FIFO buffer is included in order to provide portability, isolation, and realization of real-time disk capture and playing of ADPCM speech data. A 6802 μP controls the speech processor and FIFO buffer, and each of theses three devices are located on an external board which, in turn, is serially interfaced to the host computer. Most of the required speech processing for vocal shaping is implemented on the host computer through menu driven software. This software is capable of visually associating the waveform of speech with its sound, so that individual components may be isolated, played, duplicated, extracted, inserted, and modified (LPE).

Preliminary subjective tests were conducted in order to determine the effectiveness of the synthesis techniques, tools, and methodology for vocal shaping. Because of the unwillingness of the manufacturer (Oki) to disclose proprietary information about their ADPCM algorithm, some of the speech processing capabilities of the PC AT workstation could not be used. Instead a Macintosh IIsi speech processing workstation was used in these experiments. Nevertheless, these experiments consisted of expansion by extraction, expansion by sub-word splicing, and expansion by isolated phoneme concatenation.

Preliminary results show that new words formed by sub-word splicing (up to 80% natural quality) sounded much better than words formed by phoneme concatenation (high 30% quality). These results support the knowledge that a limited number of mono-syllabic natural sounding words can be formed by putting together extracted syllables (characterized by steady state conditions existing at both boundaries) with extracted stop consonant diphones. This result is not surprising, because the concatenation of these two types of synthesis units requires no significant consideration of coarticulation. The probable reason why these words did not achieve higher ratings is because of the 'allophonic' nature of the syllables and diphones. Much in the same way as a word sounds strange when a certain

phoneme is replaced by one of its other allophones (refer to Chapter II, Section 2.1.1.1), these words may have sounded strange because the stress, intonation, and inflection of the synthesis units were intended for the words from which they were extracted and not necessarily for the newly spliced word, which generally require different prosody.

Other preliminary results show that amplitude envelope interpolation improved the quality of the resulting word when the boundary properties of two sub-word units were modified. These preliminary results support the use of amplitude interpolation for finely adjusting the boundary properties and supplementing the coarse splicing of copy, cut, and paste for the specific words used in the test. Furthermore, this method supports sub-word extraction, so that the extracted sub-word, alone, can be used for vocal shaping.

The tests conducted on LPE of vowels show preliminarily promising results. The predicted and postdicted frames exhibited a 60% average likeness with their original sample frame.

This thesis has contributed to the general and technical knowledge through the following advances:

(1) Motivational grounds for using speech synthesis tools for automated vocal shaping systems.

(2) Optimal speech synthesis techniques, units, and tools for vocal shaping.

(3) A model of waveform concatenation using linear predictive extrapolation.

(4) Hardware and software design and implementation of a speech processing system on a PC AT.

- ◊ Implementation of a dual-pointer FIFO buffer consisting of a 6802 μP, expandable SRAMs, and dual-port I/O (serial and parallel).
- ◊ Paper design of a swinging buffer (hardware implementation).
- ◊ Implementation of serial communications at 115.2 kbps between an 80286 μP (within PC AT) and a 6802 μP (FIFO controller) through RS-232C interface.
- ◊ Menu driven software on host computer (PC AT) including disk capture and playing of ADPCM speech data, serial port initialization, time domain plot of PCM data, selection (through mouse) of any portion of speech data within a file for playing, cutting, copying, pasting, extrapolating, and averaging. Any size file may be processed since this software uses the hard disk as virtual RAM.

(5) Speech splicing methodology for vocal shaping including the use of the tools copy, cut, and paste; amplitude interpolation; and linear predictive extrapolation.

Recommendations for future work are as follows:

(1) Other fuse functions for the LPE binding segment model of coarticulation can be investigated (suggestion: use fuzzy logic to determine the contributions of abutting phonemes to the binding segment).

(2) Software for modifying pitch, stress, intonation, and inflection of isolated phonemes can be developed (suggestion: use digital filtering and equalization).

(3) Instead of using a dedicated ADPCM chip (along with the unavailable proprietary information), a straight forward PCM chip should be used for speech digitization. This can be done by multiplexing different speech digitizing chips on the existing board and writing the controller software (6802 μP instruction set) for each chip. The controller would select a certain speech digitizer and execute the corresponding program. Note that this is possible due to the modular design of the existing board. In particular, the current speech chip (MSM6258 ADPCM) is viewed as an external device and, as such, it is interfaced to the controller through a PIA on an external data bus. Because the 6258 can be placed in high impedance, this makes it possible to connect other three state speech digitizers to the external bus. With this configuration any compatible speech compression technique can be implemented through software.

REFERENCES

[Appl87] Apple Computer, *Designing Cards and Drivers for Macintosh II and Macintosh SE.* Reading MA: Addison-Wesley, 1987.

[ApMi85] Applied Microsystems Corp., *EM186 Diagnostic Emulator 6800/6802 Series Microprocessors User's Manual.* Kirkland, WA: Applied Microsystems Corp, 1985, 100 pp.

[Cair90] S. Cairns, "Computer aided speech shaping," *M.A. Thesis,* Winnipeg, Manitoba, Canada: University of Manitoba, April 1990, 77 pp.

[Camp89] J. Campbell, *The RS-232 Solution.* San Francisco CA: SYBEX Inc, 1989, 196 pp.

[Cate83] J. Cater, *Electronically speaking: Computer speech generation.* Indianapolis IN: Howard W. Sams & co. Inc, 1983, 230 pp.

[CoLa90] M. Davidson, "Get a GUI on it!," *Computer Language,* vol. 7, no.8, pp. 81-94, 1990.

[Desr90] M. Desrochers, "Computer-based versus human assessment of vocal responses with developmentally handicapped individuals," *Ph. D. Thesis.* Winnipeg, MB, Canada: University of Manitoba, 1990, 267 pp.

[FeLo89] K. Ferens and C. Love, "A speech recorder and synthesizer using ADPCM," *B. Sc. Thesis.,* Winnipeg, MB, Canada: University of Manitoba, 1989, 114 pp.

[FeKi91] K. Ferens and W. Kinsner, "Speech synthesis using fuzzy splicing," *IEEE Western Canadian Conf. Computers, Power and Communication Systems*, WESCANEX '91, Regina, SK, May 29-30, 1991 (in press).

[FlHa83] S. Fletcher and A. Hasegawa, "Speech modification by a deaf child through dynamic orometric modelling and feedback," *Journal of Speech and Hearing Disorders*, vol. 48, pp 178-85, 1983.

[JaNo84] N. Jayant and P. Noll, *Digital coding of waveforms - principles and applications to speech and video.* Englewood Cliffs, NJ: Prentice-Hall, 1984, 688 pp.

[JoCh 87] L. Jordan and B. Chuchill, "Communications and Networking for the IBM PC & Compatibles, Revised and Expanded", New York, NY: Brady, 1987, 511 pp.

[KWMR87] D. Kewley-Port, C. Watson, D. Maki, and D. Reed, "Speaker-dependent speech recognition as the basis for a speech training aid", *Proc. of IEEE Neural Networks*, Cat. no. 87 CH2396-0, pp. 372-375, 1987.

[Kins88] W. Kinsner, "Microprocessor and Microcomputer Interfacing for Real-Time Systems," *Lecture Notes*, Winnipeg, MB, Canada: Department of Electrical and Computer Engineering, University of Manitoba: Winnipeg, MB, Canada, 1988.

[Klim87] G. Klimenko, "A study of ADPCM, CVSD, and phoneme speech coding techniques," *M.Sc. Thesis*. Winnipeg, MB, Canada: University of Manitoba, 1987.

[KlKi87] G. Klimenko and W. Kinsner, "A study of ADPCM, CVSD, and PSS speech coding techniques," *Proc. IEEE Engineering in Medicine and Biology Soc.*, IEEE Cat. no. 87 CH2513-0, pp.1797-98, 1987.

[LoFK89] C. Love, K. Ferens, and W. Kinsner, "A speech recorder and synthesizer using ADPCM," *Proc. IEEE Engineering in Medicine and Biology Soc.*, IEEE Cat. no. 89 CH2770-6, pp. 659-60, 1989.

[LoKi90] C. Love and W. Kinsner, "A phonemic recognizer for speech therapy using a neural network model," *Proc. Canadian Medical and Biological Engineering Soc.*, CMBS Cat. no. 90 CMBC-16-CCGB, pp. 93-4, 1990.

[MacU90] R. Myslewski, "Three Cheers for Three New Macs," *MacUser Labs*, pp. 90-111, , 1990.

[Mono85] *LSI Databook*. Santa Clara, CA: Monolithic Memories inc, 1985.

[MSQC88] Microsoft, *C For Yourself*. Redmond, WA: Microsoft Corp, 1988, 376 pp.

[NaSe89] National Semicondictor, *Advanced Peripherals Data Communications Local Area Networks UARTs Handbook*. Santa Clara, CA : National Semiconductor Corp , 1988.

[NoWi85] P. Norton and R. Wilton, *The New Peter Norton Programmer's Guide to the IBM PC & PS/2*. Redmond, WA: Microsoft Press, 1985 500 pp.

[OkiS90] Oki Semiconductor, *Oki Voice Synthesis LSI 1990*. Sunnyvale, CA: Oki Semiconductor, 1990, 338 pp.

[Pars87] T. Parsons, *Voice and Speech Processing*. New York, NY: McGraw Hill, 1987, 402 pp.

[Perk71] W. H. Perkins, *Vocal function: Assessment and therapy, Handbook of speech pathology and audiology*, Prentice-Hall: Englewood Cliffs, NJ, pp. 505-34, 1971.

[Pete88] D. Peters, "A speech recognizer using LPC and DTW," *B.Sc. Thesis*. Winnipeg, MB, Canada: University of Manitoba, 1988.

[PeKR87] J.J. Pear, W. Kinsner, and D. Roy, "Vocal shaping of retarded and autistic individuals using speech synthesis and recognition," *Proc. IEEE Engineering in Medicine and Biology Soc.*, IEEE Cat. no. 87 CH2513-0, pp. 1787-88, 1987.

[RaDu86] R. Duncan, *Advanced MSDOS Programming*. Redmond, WA: Microsoft Press, 1986, 669 pp.

[RaSc78] L. Rabiner and R. Schafer, *Digital processing of speech signals*. Englewood Cliffs, NJ: Prentice Hall, 1978, 512 pp.

[RCA84] RCA, *CMOS Microprocessors, Memories and Peripherals Databook*. USA: RCA Corp, 1984.

[MaPe88] G. Martin and J. Pear, *Behavior modification: What it is and how to do it (3rd Edition)*. Englwood Cliffs, NJ: Prentice-Hall, 1988.

[Skin53] B. Skinner, *Science and Human Behaviour*. New York, NY: Macmillan, 1953.

[SkSh78] P. Skinner and R. Shelton, *Speech, language, and hearing: Normal processes and disorders*. Reading, Mass: Addison-Wesley, 1978.

[Sagi90] Y. Sagisaka, "Speech Synthesis from Text," *IEEE Comunnications Magazine* 0163-6804, vol. 1, pp 35-41, 1990.

[Swan87] C. Swanson, "A study and implementation of real-time linear predictive coding of speech," *M.Sc. Thesis*. Winnipeg, MB, Canada: University of Manitoba, 1987, 228 pp.

[Texa84] Texas Instruments, *The TTL Data Book 1984*. Dallas, Texas: Texas Instruments inc, 1984.

[WeDi83] Western Digital, *Western Digital 1983 Components Handbook*. Irvine, CA: Western Digital corp, 1983.

[Wier82] A. Wierach, "A study of English stop consonant synthesis and perception," *M.Sc. Thesis*. Winnipeg, MB, Canada: University of Manitoba, 1982, 126 pp.

APPENDIX A: SOFTWARE LISTING

A.0 Introduction

This appendix describes and lists the source code of the Memory Manager and Host computer.

A.1 Memory Manager Software

This section describes and lists the Memory Manager software. It is written in 6802 assembly language. The following source code is augmented with detailed comments. Only the comments appearing inline with the code or prefixed with a "*" are assembable. In other words, to assemble the following code, remove the comments appearing in paragraph form.

A.1.1 Main

```
ACIA_IO_Reg     equ     $1D01
ACIA_CR_Stat    equ     $1D00
PIA_PortA_Reg   equ     $1E00
PIA_DDRA        equ     $1E00
PIA_CRA         equ     $1E01
PIA_PortB_Reg   equ     $1E02
PIA_DDRB        equ     $1E02
PIA_CRB         equ     $1E03
CR              equ     $0D
LF              equ     $0A
BuffC_Addr      equ     $6000
BuffB_Addr      equ     $4000
BuffA_Addr      equ     $2000
                org     $F800
Read_Ptr        rmb     2
Write_Ptr       rmb     2
rts_Status      rmb     1
Seconds         rmb     1

StartPgm        sei                     *Interrupt mask bit is set to temporarily
                                        *disable all interrupts.
                lds     #127            *The stack pointer is loaded with the
                                        *starting address of the stack. Stack
                                        *grows downward.
                jsr     VariableInit    *Initialize several system variables.
                jsr     PIAInit         *Initialize the parallel port PIA.
```

```
                jsr     ACIAInit        *Initialize the serial port ACIA.
                jsr     GetCommand      *Get a command from Host.
                cmpa    #$02            *GetCommand returned a Host issued
                                        *command in accumulator A.
                beq     DoRecord        *02₁₆ = record 2 seconds.
                cmpa    #$04
                beq     DoRecord        *04₁₆ = record 4 seconds.
                cmpa    #$10
                beq     DoRecord        *10₁₆ = record 16 seconds.
                cmpa    #$20
                beq     DoRecord        *20₁₆ = record 32 seconds.
                cmpa    #$AA
                beq     DoPlayback      *AA16 = code for playback command.
                bra     StartPgm

DoRecord        lsra
                staa    Seconds
                jmp     Record
DoPlayback      jmp     Playback
```

A.1.2 Record

The following group of code is the record routine, and it consist of five segments, command acknowledge, initialization subroutines, transmit speech data in the foreground, record speech data in the background, and stop record and empty buffer.

Segment One: Command Acknowledge

```
Record      ldaa    #$55            *B6=1 sets RTSB=1, and this informs
            staa    ACIA_CR_Stat    *the Host that the Slave received a valid
                                    *command.

WaitForAck  ldaa    ACIA_CR_Stat    *Wait for Host acknowledge
            anda    #$08            *B3=0 means Host is still trying to send
                                    *a command.
            beq     WaitForAck      *B3=1 means acknowledge.
```

Segment Two: Intitialization Subroutines

```
            jsr     InstallRecAddr  *Install record interrupt routine address.
            jsr     RecordInit      *Initialize the speech chip for record.
```

Segment Three: Transmit Foreground

The following segment of code is the transmit foreground routine. In this routine the controller continually transmits speech data to the Host via the serial interface. Speech data transmitted in this way is read from the buffer at location pointed to by Read_Ptr. While executing this routine, the controller is interrupted every 250 μsec by the speech chip. This causes the controller to execute a background routine, during which speech data is read from the speech chip and written to the buffer. Since the process of transmiting the buffer is about twice as fast as writing the buffer, at some time the buffer becomes empty. The controller checks for equality of the pointers, i.e., Read_Ptr = Write_Ptr. If true, the buffer is empty and transmitting is temporarily suspended, until the pointers are no longer equal, which becomes true after the speech chip interrupts the foreground, and a byte is read from the speech chip and written to the buffer.

```
SendData    cli                     *Enable the 6802 to respond to an
                                    *interrupt. Interrupts are requested by
                                    *the speech chip.

ReqToSend   ldaa    #$15            *B6=0 resets RTSB=1, and this informs
            staa    ACIA_CR_Stat    *the Host that the Slave is requesting to
                                    *send.
```

```
WaitForCTS0     ldaa    ACIA_CR_Stat    *Determine whether Host is ready to
                anda    #$08            *receive speech data.  Wait until ready.
                bne     WaitForCTS0     *B3=1 indicates negative, i.e., No.
                                        *B3=0 indicates positive, i.e., Yes.

TxDNotEmpty     ldaa    ACIA_CR_Stat    *Check the Transmit Data Register
                rora                    *Empty (TDRE) bit.  Rotate TDRE into
                rora                    *carry bit of condition code register.
                bcc     TxDNotEmpty     *C=0 indicates TxD is not empty.

                ldx     Read_Ptr        *Read_Ptr points to next byte in buffer.

Wait0           cpx     Write_Ptr       *Check whether the buffer is empty.
                beq     Wait0           *Buffer is empty if there are no data to
                                        *read, i.e., if Read_Ptr = Write_Ptr.

                ldaa    0,X             *Read byte at location pointed to by
                staa    ACIA_IO_Reg     *Read_Ptr and transmit to Host.

                inx                     *Read_Ptr points to the next data byte.
                stx     Read_Ptr        *Store Read_Ptr in anticipation for next
                                        *transmission.

                cpx     #BuffC_Addr     *Determine whether Write_Ptr is
                                        *pointing to the last location + 1.
                beq     ResetReadPtrRec *If true, initialize Write_Ptr to point to
                                        *the first location of the FIFO buffer.

                jmp     WaitForCTS0     *Continue receiving data from the Host.

ResetReadPtrRec ldx     #BuffA_Addr     *Write_Ptr now point to location
                                        *2000_16, which is the starting address
                                        *of SRAM1, and the virtual start of the
                                        *circular FIFO buffer.
                stx     Read_Ptr        *Save current write location.

                jmp     WaitForCTS0     *Continue receiving data from the Host.
```

Segment Four: Record Background

The following segment of code is the record interrupt routine, which is processed in the background. This routine is executed each time the speech processor indicates that the next byte of ADPCM data is ready for reading. The interrupt signal is generated every 250 μsec on control line CA1, which is input to the PIA. The PIA relays this interrupt signal to the μP via the IRQ line. The starting address, i.e., IRQRecStart, is loaded into emulator RAM at location $fff8_{16}$ to $fff9_{16}$ during record initialization subroutines. IRQRecStart reads a byte of speech data from the speech chip and writes the byte to the FIFO buffer at location pointed to by Write_Ptr . This routine also checks for the stop condition by decrementing the seconds count by one each time an 8K buffer becomes full of speech data.

```
IRQRecStart     ldx     Write_Ptr       *Load the Write_Ptr with the next
                                        *available and empty location.
                ldaa    #$10            *This delay is incorporated in order to
                jsr     Delay           *find the subjectively optimum time to
                                        *read from the speech chip.

                ldab    #$85            *B7=1 and B6=0 used by BRG.
                                        *B3=0 enables the speech chip.
                                        *B2=1 selects the data mode.
                                        *B1=0 puts the read line (RD) low.
                stab    PIA_PortB_Reg   *B0=1 disables write operation.

                ldaa    PIA_PortA_Reg   *Read a byte from the speech chip.
                staa    0,X             *Store byte in buffer at location pointed
                                        *to by Write_Ptr.
                ldaa    #$87            *B1=1 puts the read line (RD) high.
                staa    PIA_PortB_Reg

                inx                     *Point to the next empty location.
                cpx     #BuffB_Addr     *Determine whether Write_Ptr is
                                        *pointing to starting address of BufferB.
                beq     DecSeconds      *If so, decrement seconds counter.

CheckRollOver   cpx     #BuffC_Addr     *Determine whether Write_Ptr is
                beq     ResetWritePtrRec *pointing to the last location + 1.

                stx     Write_Ptr       *Store pointer for next interrupt.
                ldaa    PIA_PortA_Reg   *Clear the interrupt flag of the PIA.
                rti                     *Return to foreground processing.

DecSeconds      dec     Seconds         *8K of SRAM = 2 sec of speech.
                beq     StopRecord      *
```

```
                bra     CheckRollOver       *

ResetWritePtrRec ldx    #BuffA_Addr         *Write_Ptr now point to the virtual start
                                            *of the circular FIFO buffer, i.e.,
                                            *2000₁₆, the starting address of
                stx     Write_Ptr           *SRAM1.

                dec     Seconds
                beq     StopRecord
                ldaa    PIA_PortA_Reg       *Clear the interrupt flag of the PIA.
                rti                         *Return to foreground processing.
```

Segment Five: Stop Record and Empty Buffer

The following segment of code stops the record mode of the speech chip and transmits the remaining speech data from the buffer to the Host. The empty part of this code is required because, after the record time has elapsed, there may be some speech data in the buffer not yet transmitted to the Host.

```
StopRecord      jsr     Stop

WaitForCTS1     ldaa    ACIA_CR_Stat        *Determine whether Host is ready to
                anda    #$08                *receive speech data. Wait until ready.
                bne     WaitForCTS1         *B3=1 indicates negative, i.e., No.
                                            *B3=0 indicates positive, i.e., Yes.

TxDNotEmpty1    ldaa    ACIA_CR_Stat        *Check the Transmit Data Register
                rora                        *Empty (TDRE) bit. Rotate TDRE into
                rora                        *carry bit of condition code register.
                bcc     TxDNotEmpty1        *C=0 indicates TxD is not empty.

                ldx     Read_Ptr            *Read byte from buffer pointed to by
                ldaa    0,X                 *Read_Ptr and transmit to Host.
                staa    ACIA_IO_REG         *
                inx
                cpx     #BuffB_Addr         *Is the Buffer empty yet?
                beq     Finished            *If yes, goto Finished routine.
                cpx     #BuffC_Addr         *Is the Buffer empty yet?
                bne     WaitForCTS1         *If not continue transmitting buffer.

Finished        ldaa    #$55                *RTSB=1 informs the Host that the
```

```
staa     ACIA_CR_STAT      *slave is finished recording.
jmp      StartPgm          *Jump to the start of the program.
```

Note that because this branch is part of the interrupt routine, the context of the machine prior to execution of the interrupt routine is still located on the stack and the stack pointer is pointing to the next available location. However, this does not present a problem, since the controller jumps to the start of the program where the system is initialized. In particular, the stack pointer is reset to the starting of the stack, i.e., location 127. Therefore, the current contents of the stack are discarded.

A.1.3 Playback Routine

The following group of code is the playback routine, and it consist of four segments, command acknowledge, initialization subroutines, receive speech data in the foreground, and playback speech data in the background.

Segment One: Command Acknowledge

```
Playback        ldaa    #$55            *B6=1 sets RTSB=1, and this informs
                staa    ACIA_CR_Stat    *the Host that the Slave received a valid
                                        *command.

WaitForAck1     ldaa    ACIA_CR_Stat    *Wait for Host acknowledge
                anda    #$08            *B3=0 means Host is still trying to send
                                        *a command.
                beq     WaitForAck1     *B3=1 means acknowledge.
```

Segment Two: Intitialization Subroutines

```
                jsr     FillQueue       *Get 256 bytes of speech data.
                jsr     InstallPlayAddr *Install play interrupt routine address.
                jsr     PlayBackInit    *Initialize the speech chip for playback.
```

Segment Three: Receive Foreground

The following segment of code is the foreground routine. In this routine the controller continually receives speech data from the Host via the serial interface. Speech data received in this way is stored in the buffer at location pointed to by Write_Ptr. While executing this routine, the controller is interrupted every 250 μsec by the speech chip. This causes the controller to execute a background routine, during which one byte of speech data is read from the buffer and written to the speech chip. Since the process of loading the buffer is about twice as fast as reading the buffer, at some time the buffer becomes full. The controller checks for equality of the pointers, i.e., Write_Ptr = Read_Ptr. If true, the buffer is full and loading is temporarily suspended, until the pointers are no longer equal, which becomes true after the speech chip subsequently interrupts the foreground and a byte is read from the buffer.

```
FetchData       cli                     *Enable the 6802 to respond to an
                                        *interrupt. Interrupts are requested by
                                        *the speech chip.

WaitForHost1    ldaa    ACIA_CR_Stat    *Determine whether Host is
                anda    #$08            *requesting to send speech data.
```

```
                bne     WaitForHost1        *B3=1 indicates negative, i.e., No.
                                            *B3=0 indicates positive, i.e., Yes.

Wait1           ldx     Write_Ptr           *Load x with the next available location
                                            *in the buffer.
                cpx     Read_Ptr            *Determine whether the buffer is full. If
                bne     LoadBuffer          *true, temporarily suspend receiving
                                            *data from Host.

                ldab    #$55                *B6=1 sets RTSB=1 and this informs
                stab    ACIA_CR_Stat        *the Host not to send data.
                stab    rts_Status          *Preserve the current state of RTSB
                                            *The current state is used by the
                                            *interrupt routine.
                jmp     Wait1               *Temporarily suspend loading buffer
                                            *until Write_Ptr ≠ Read_Ptr.

LoadBuffer      ldaa    #$15                *Host is requesting to send and the
                staa    ACIA_CR_Stat        *buffer is not full. Therefore, give the
                                            *Host clearance to send. B6=0 resets
                                            *RTSB=0.
                staa    rts_Status          *Preserve the current state of RTSB.

RxDEmpty0       ldaa    ACIA_CR_Stat        *Determine whether the ACIA receive
                rora                        *data register (RxD) is full.
                bcc     RxDEmpty0           *C=0 indicates RxD is currently empty.
                ldaa    ACIA_IO_Reg         *Receive data from the Host.
                staa    0,x                 *Store data at the location pointed to by
                                            *Write_Ptr.
                inx                         *Point to the next available location.
                stx     Write_Ptr           *Save current write location.

                cpx     #BuffC_Addr         *Determine whether Write_Ptr is
                                            *pointing to the last location + 1.
                beq     ResetWritePtrPlay   *If true, initialize Write_Ptr to point to
                                            *the first location of the FIFO buffer.

                jmp     Wait1               *Continue receiving data from the Host.

ResetWritePtrPlay ldx   #BuffA_Addr         *Write_Ptr now points to location
                                            *2000_16, which is the starting address
```

```
                                        *of SRAM1, and the virtual start of the
                                        *circular FIFO buffer.
            stx     Write_Ptr           *Save current write location.

            jmp     Wait1               *Continue receiving data from the Host.
```

Segment Four: Playback Background

The following group of code is the playback interrupt routine, which is processed in the background. This routine is executed each time the speech processor is ready for the next byte of ADPCM data. The interrupt signal is generated every 250 μsec on control line CA1, which is input to the PIA. The PIA relays this interrupt signal to the μP via the IRQ line. The starting address, i.e., IRQPlayStart, is loaded into emulator RAM at location $fff8_{16}$ to $fff9_{16}$ during playback initialization subroutine. This routine reads a byte of speech data from the FIFO buffer pointed to by Read_Ptr and writes this byte to the speech chip. This routine also checks for the buffer empty condition, which is implied by the condition when the Read_Ptr points to the same location as the Write_Ptr. If empty condition is true, the controller branches to the stop routine and playback is stopped.

```
IRQPlayStart    ldaa    #$55            *(2)B6=1 sets RTSB=1 in order to
                                        *inform the Host not to send because the
                                        *Slave is busy servicing the speech
                staa    ACIA_CR_Stat    *(5)chip.

ReadBuffer      ldx     Read_Ptr        *(5)Load the Read_Ptr with the location
                                        *containing the next ADPCM byte.
                ldaa    0,X             *(5)Read byte pointed to by Read_Ptr.
                staa    PIA_PortA_Reg   *(5)Place byte on speech chip data bus.

                ldab    #$86            *(2)
                stab    PIA_PortB_Reg   *(5)B0=0 write line (WR) low.
                ldaa    #$87            *(2)B0=1 write line (WR) high, and this
                staa    PIA_PortB_Reg   *(5)writes speech data to speech chip.
```

Finish current instruction time:	5 μsec.	
Interrupt entrance time:		13 μsec.
Read and write time:	2+5+5+5+5+2+5+2+5=36 μsec.	

Total time required before speech data is actually written to speech chip: 54 μsec.

```
                inx                     *Point to next ADPCM byte.
```

```
                cpx     Write_Ptr       *Determine whether buffer is empty.
                beq     StopPlayback    *If true, terminate playback routine.

                cpx     #BuffC_Addr     *Determine whether Read_Ptr is
                                        *pointing to the last location + 1.
                beq     ResetReadPtrPlay *If true, initialize Read_Ptr to point to
                                        *the first location of the FIFO buffer.

                stx     Read_Ptr        *Store Read_Ptr in anticipation of the
                                        *next interrupt routine
                ldaa    rts_Status      *Return to foreground with the state of
                                        *RTSB prior to execution of the
                staa    ACIA_CR_Stat    *interrupt routine.
                ldaa    PIA_PortA_Reg   *The data register (Port A) of the PIA
                                        *side that is used to interrupt the µP
                                        *must be read in order to reset the
                rti                     *interrupt flag of the PIA.

ResetReadPtrPlay ldx    #BuffA_Addr     *Read_Ptr now points to the virtual start
                                        *of the circular FIFO buffer, i.e.,
                                        *2000₁₆, the starting address of
                stx     Read_Ptr        *SRAM1.

                ldaa x  rts_Status      *This is the same return code as above.
                staa    ACIA_CR_Stat    *It is repeated in this way in order to
                ldaa    PIA_PortA_Reg   *save time.
                rti                     *ReTurn from Interrupt (RTI).

StopPlayback    jsr     Stop
                jmp     StartPgm        *Jump to the start of the program.
```

Note that because this branch is part of the interrupt routine, the context of the machine prior to execution of the interrupt routine is still located on the stack and the stack pointer is pointing to the next available location. However, this does not present a problem, since the controller jumps to the start of the program where the system is initialized. In particular, the stack pointer is reset to the starting of the stack, i.e., location 127. Therefore, the current contents of the stack are discarded.

A.1.4 Subroutines

A.1.4.1 FillQueue

This subroutine fills the playback buffer with 256 bytes of speech data. The Host transmits these data via the RS-232C interface, and the controller reads these data through the ACIA. This subroutine is called by the playback routine. Filling the queue with 256 bytes requires about 25 msec, and this is perceptually unnoticed.

```
FillQueue       nop
WaitForHost     ldaa    ACIA_CR_Stat    *Check if Host is requesting to send.
                anda    #$08            *B3=1 means Host is not requesting to
                bne     WaitForHost     *send.  B3 = 0 means Host is rts.

ReceiveMore     ldaa    #$15            *Given that the Host is requesting to send, the
                staa    ACIA_CR_Stat    *Slave now gives the Host clearance to send.

RxDRegEmpty     ldaa    ACIA_CR_Stat    *Determine the state of the Receive Data
                rora                    *Register (RxD).  Rotate B0 into carry.
                bcc     RxDRegEmpty     *B0=1 means RxD is full.

                ldab    #$55            *Having received a byte, the Slave
                stab    ACIA_CR_Stat    *acknowledges this by setting RTSB=1,
                                        *informing the Host not to send.

ReadByte        ldaa    ACIA_IO_Reg     *Read the byte from ACIA register.
                ldx     Write_Ptr       *Load x with the address of the next
                                        *available location within the FIF0.
                staa    0,X             *Store the byte at this location.
                inx                     *Increment the write pointer, and store
                stx     Write_Ptr       *it in anticipation of the next byte.

                cpx     #$2100          *Check if done receiving 256 bytes.
                bne     ReceiveMore     *If not, get some more

QueueFull       stab    rts_Status      *The status of RTSB is saved in anticipation of
                                        *the next routine, which may be interrupted
                                        *before it can itself define and save the state of
                                        *RTSB.The queue is full and contains 256 bytes
                                        *of speech data.
                rts                     *Execute return from subroutine.
```

A.1.4.2 PlayBackInit

This subroutine initializes the speech chip for playback. The PIA is first set up to write the playback command to the speech chip. Next the playback command is written. And finally, the PIA is configured to enable interrupts via positive edge CA1.

```
PlayBackInit    clra
                staa    PIA_CRA         *Reset Control Register A (CRA)
                staa    PIA_CRB         *Reset Control Register B (CRB)
                ldaa    #$FF            *Configure all 8 bits of both Port A and
                staa    PIA_DDRB        *Port B as output data registers.  B0 to
                                        *B3 of Port B are used supply the
                                        *control signals (WR, RD, D/C, and
                                        *CS, respectively) to the speech chip.
                staa    PIA_DDRA        *Port A is used to write commands and
                                        *speech data to the speech chip.
                ldaa    #$04            *B2=1 Selects Port A data register
                staa    PIA_CRA         *Disable interrupts via CA1 and CA2.
                staa    PIA_CRB         *B2=1 Selects Port B data register.
                                        *Disable interrupts via CB1 and CB2.

                ldaa    #$02            *02(16) = code for playback command.
                staa    PIA_PortA_Reg   *Place playback command on data bus.

                ldaa    #$82            *B7=1 and B6=0 used by BRG.
                staa    PIA_PortB_Reg   *CS=B3=0 Enables speech chip.
                                        *D/C=B2=0 specifies command input.
                                        *RD=B1=1 disables read operation.
                                        *WR=B0=0 enables write operation.
                ldaa    #$87            *D/C=B3=1 specifies data I/O.
                staa    PIA_PortB_Reg   *WR=B0=1 toggles WR, and command
                                        *is latched into the speech chip.

                ldaa    #$07            *B2=1 Selects Port A data register.
                staa    PIA_CRA         *B1=1 B1=0 defines CA1 active
                                        *transition as low to high.  B0=1
                                        *enables interrupts via active transitions
                rts                     *of CA1.  Return to calling program.
```

A.1.4.3 RecordInit

This subroutine initializes the speech chip for record. The PIA is configured to communicate with the speech chip in the record mode, and the record command is written to the speech chip. The PIA is first set up to write the record command to the speech chip. Next the record command is written. And finally, the PIA is configured to enable interrupts via negative edge CA1.

```
RecordInit      clra
                staa    PIA_CRA         *Reset Control Register A (CRA)
                staa    PIA_CRB         *Reset Control Register B (CRB)
                ldaa    #$FF            *Configure all 8 bits of both Port A and
                staa    PIA_DDRB        *Port B as output data registers.  B0 to
                                        *B3 of Port B are used to supply the
                                        *control signals (WR, RD, D/C, and
                                        *CS, respectively) to the speech chip.
                staa    PIA_DDRA        *Port A is used to write commands and
                                        *speech data to the speech chip.
                ldaa    #$04            *B2=1 Selects Port A data register.
                staa    PIA_CRA         *Disable interrupts via CA1 and CA2.
                staa    PIA_CRB         *Disable interrupts via CB1 and CB2.

                ldaa    #$04            *04(16) = code for record command.
                staa    PIA_PortA_Reg   *Place record command on data bus.

                ldaa    #$82            *B7=1 and B6=0 used by BRG.
                staa    PIA_PortB_Reg   *CS=B3=0 Enables speech chip.
                                        *D/C=B2=0 specifies command input.
                                        *RD=B1=1 disables read operation.
                                        *WR=B0=0 enables write operation.
                ldaa    #$87            *D/C=B3=1 specifies data I/O.
                staa    PIA_PortB_Reg   *WR=B0=1 toggles WR, and command
                                        *is latched into the speech chip.

                clra                    *Configure all 8 bits of Port A as input.
                staa    PIA_CRA         *
                staa    PIA_DDRA        *
                ldaa    #$05            *B2=1 selects Port A data register.
                staa    PIA_CRA         *B1=0 defines CA1 active edge as
                                        *high to low or negative edge.
                rts                     *ReTurn form Subroutine (RTS).
```

A.1.4.4 ACIAInit

This subroutine initializes the ACIA. The data format is selected as follows: 115.2kbps, 8, N, 1. The receiver and transmitter interrupts are disabled. RTSB is set to logic 1.

```
ACIAInit    ldaa    #43             *Master reset the ACIA and define the
            staa    ACIA_CR_Stat    *state of RTSB=1, inactive.
            ldaa    #$10            *Delay in order to allow ACIA reset.
            jsr     Delay           *Not really required, but good idea.
            ldaa    #$80            *B7 and B6 are connected to the inputs
                                    *of the Bit Rate Generator
                                    *(BRG). B7=1 and B6=0 select the
                                    *multiply by 16 function. The ACIA
                                    *RxCLK and TxCLK are fed by pin 17
                                    *of the BRG. A 1.8432 MHz clock is
                                    *output from this pin, since the BRG is
                                    *selected to multiply by 16 and pin 17
                                    *selects the 115.2 kHz clock,
            staa    PIA_PortB_Reg   *i.e., (16)115.2 = 1.8432 MHz.
            ldaa    #$55            *Data format configuration: 115.2 kbps,
                                    *8 bits, no parity, and 1 stop bit.
            staa    ACIA_CR_Stat    *Disable interrrupts and RTSB=1.
            rts                     *ReTurn from Subroutine (RTS).
```

A.1.4.5 PIAInit

This subroutine initializes the PIA. Port A is used to supply the speech chip with the control signals, CS, D/C, RD, and WR. Port B is used to communicate commands, status, and speech data with the speech chip. The PIA control line CA1 is used by the speech chip in order to interrupt the μP. However, interrupts are disabled until the actual record or playback routine is executed, where the active edge of CA1 is known.

```
PIAInit     clra
            staa    PIA_CRA          *Access PIA A data direction register.
            staa    PIA_CRB          *Access PIA B data direction register.
            ldaa    #$ff
            staa    PIA_DDRA         *Port A configured for output.
            staa    PIA_DDRB         *Port B configured for output.
            ldaa    #$04             *Access data register of Port B.
            staa    PIA_CRA          *Disable interrupts via CA1 and CA2.
            staa    PIA_CRB          *Disable interrupts via CB1 and CB2.
            ldaa    #$8f             *B7=1 & B6=0 used by BRG.
            staa    PIA_PortB_Reg    *B0=WR=1 disables Write operation.
                                     *B1=RD=1 disables Read operation.
                                     *B2=D/C=1 selects data mode.
                                     *B3=CS=1 disables speech data bus.
            rts                      *ReTurn from Subroutine (RTS).
```

A.1.4.6 Miscllaneaous

Subroutine VariableInit

This subroutine initializes several system variables.

```
VariableInit    clra
                staa    rts_Status      *Stores the status of RTSB.
                ldx     #BuffA_Addr     *Initialize the read and write pointers to
                                        *point at the same location, i.e., 2000₁₆,
                stx     Read_Ptr        *which is the starting lcoation of
                stx     Write_Ptr       *SRAM1.
                rts                     *ReTurn from Subroutine (RTS).
```

Subroutine InstallPlayAddr

This subroutine installs the address of the interrupt routine for playback. The address IRQPlayStart is loaded into emulator RAM at location $fff8_{16}$ to $fff9_{16}$. Normally, when the software has been debugged and developed, the system firmware, including the interrupt addresses, is burnt into EPROM and cannot be changed by software. However, while debugging and developing the code, the interrupt address is loaded into emulator RAM, which is the emulator's version of system ROM. In this way, all system software, including reset address, nonmaskable interrupt address, interrupt address, etc. can be easily modified.

```
InstallPlayAddr ldx     #IRQPlayStart   *IRQPlayStart is a 16 bit address.
                stx     $FFF8           *High order 8 bits loaded at fff8₁₆.
                                        *Low order 8 bits loaded at fff9₁₆.
                rts                     *ReTurn from Subroutine (RTS).
```

Subroutine InstallRecAddr

This subroutine installs the address of the interrupt routine for record. The address IRQRecStart is loaded into emulator RAM at location $fff8_{16}$ to $fff9_{16}$.

```
InstallRecAddr  ldx     #IRQRecStart    *IRQRecStart is a 16 bit address.
                stx     $FFF8           *High order 8 bits loaded at fff8₁₆.
                                        *Low order 8 bits loaded at fff9₁₆.
                rts                     *ReTurn from Subroutine (RTS).
```

Subroutine Delay

This subroutine provides a time delay. The routine expects the number of times to loop to be contained in accumulator A. One pass through the loop requires 6 µsec.

```
Delay       deca                *Accumulator A - 1.
            bne     Delay       *Loop until accumulator A is zero.
            rts                 *ReTurn from Subroutine (RTS).
```

Subroutine GetCommand

This subroutine receives a command that is sent by the Host computer. GetCommand returns to the calling program with the command code in accumulator A.

```
GetCommand  nop
WaiTForRTS0 ldaa    ACIA_CR_Stat    *Wait until Host indicates a request to
            anda    #$08            *send.
                                    *B3=0 indicates request to send (RTS).
            bne     WaiTForRTS0     *B3=1 indicates no request.

            ldaa    #$15            *Give the Host clearance to send.
            staa    ACIA_CR_Stat    *B6=0 resets RTSB=0.

RxDNotFull  ldaa    ACIA_CR_Stat    *Test the received data register full
            rora                    *(RDRF) bit. Rotate RDRF into carry
            bcc     RxDNotFull      *C=0 indicates RxD not full.
            ldaa    ACIA_IO_Reg     *Read byte received form Host.
            rts                     *ReTurn from Subroutine (RTS).
```

A.1.4.7 Stop

This routine stops the current process of the speech chip. The PIA is configured in order that a stop command be written to the speech chip. Either record or playback is stopped, and the speech chip data bus is placed on high impedance.

```
Stop    clra
        staa    PIA_CRA         *Reset Control Register A (CRA)
        staa    PIA_CRB         *Reset Control Register B (CRB)
        ldaa    #$FF            *Configure all 8 bits of both Port A and
        staa    PIA_DDRB        *Port B as output data registers.  B0 to
                                *B3 of Port B are used to supply the
                                *control signals (WR, RD, D/C, and
                                *CS, respectively) to the speech chip
        staa    PIA_DDRA        *Port A is used to write commands and
                                *speech data to the speech chip.
        ldaa    #$04            *B2=1 Selects Port A data register.
        staa    PIA_CRA         *B0=0 disables interrupts via active
                                *transition of CA1 and CA2.
        staa    PIA_CRB         *B0=0 disables interrupts via active
                                *transition of CB1 and CB2.

        ldaa    #$01            *01(16) is the code for stop command.
        staa    PIA_PortA_Reg   *Place stop command on data bus.
        ldaa    #$82            *B7=1 and B6=0 used by BRG.
        staa    PIA_PortB_Reg   *CS=B3=0 enables speech chip.
                                *D/C=B2=0 specifies command input.
                                *RD=B1=1 disables read operation.
                                *WR=B0=0 enables write operation.
        ldaa    #$83            *
        staa    PIA_PortB_Reg   *WR=B0=1 toggles WR, and command
                                *is latched into the speech chip.
        ldaa    #$8f            *CS=B3=1 disables speech chip.
        staa    PIA_PortB_Reg   *

        ldaa    PIA_PortA_Reg   *Dummy reads of Port A and Port B.
        ldaa    PIA_PortB_Reg
        rts

        end                     *Assembler directive indicating the end
                                *of code.
```

A.2 Host Software

This section describes and lists the Host computer software. It is written in 8086 assembly language and Microsoft Quick C high level language. The following source code is augmented with detailed comments. Compileable code is contained within horizontal line separators. Compileable comments are contained within delimiters as follows: /* Comment */.

A.2.1 Main

```
#include <conio.h>
#include <string.h>
#include <graph.h>
#include <stdio.h>
#include <stddef.h>
#include <ctype.h>
#include <bios.h>
#include <menu.h>
#include <fprotype.h>
#include <stdlib.h>
/* Default menu attribute. The default works for color or B&W. You can override the default value by
defining your own MENU variable and assigning it to mnuAtrib, or you can modify specific fields at run
time. For example, you could use a different attribute for color than for black and white.  */

struct MENU mnuAtrib =
{
    _TBLACK, _TBLACK, _TWHITE, _TBRIGHTWHITE, _TBRIGHTWHITE,  _TWHITE,
    _TWHITE, _TBLACK, _TWHITE, _TBLACK, FALSE,
    '/', 'ø', 'Ÿ', '¿', '≥', 'ƒ'
};

struct ITEM aItem[] =
{
                    /*  Highlight Char      Pos   */
    2, "  File",    /*        Q             2     */
    0, "Display",   /*        C             0     */
    0, "Splice",    /*        R             0     */
    0, "Library",   /*        T             0     */
    0, "Assemble",  /*        S             0     */
    0,      NULL
};
```

```
/*                                  MAIN PROGRAM                                */
void  main()
{
        unsigned uKey;                          /* Unsigned key code           */
        int Flag = 1;
        while( Flag != 0 )          /* Flag = 0 means User selected Quit from menu */
        {
                ShowMainMenu();
                uKey = GetControlKey( WAIT );   /* Wait until uKey is pressed. */
                switch( uKey )                  /* Evaluate uKey               */
                {
                        case ALT:       Flag = ChooseFromMenu();
                        break;
                }
        }
        end_program();
}                                               /* End of main program.        */

/*                                  FUNCTIONS                                   */
int  Mainmenu()
{
int ret1,Flag,r1,r2,c1,c2,lns,att;
struct ITEM  m1[]=
{
                                /* Highlight Char       Pos     */
        0, "Initialize",        /*      I               0       */
        0, "Record",            /*      R               0       */
        0, "Playback",          /*      P               0       */
        0, "Quit",              /*      Q               0       */
        0, NULL
};
ClearBox(2,2,8,15,6,1);
ret1 = Menu(2, 1, m1, 0);
switch(ret1)
{
        case ESC: return ESC;
        case U_RT:      ClearTextWindow( 2,0,12,17,_TBLUE ); return U_RT;
        case U_LT:      ClearTextWindow( 2,0,12,17,_TBLUE ); return U_LT;
        case 0 :        submenuinit();     return ESC;
        case 1 :        submenurecord();   return ESC;
        case 2 :        Flag = submenuplayback(); return ESC;
```

```
        case 3 :        Flag = 0; return Flag; break;
}
}                                       /* End of function MainMenu.                   */


int  Displaymenu( void )
{
int ret1,Flag,r1,r2,c1,c2,lns,att;
struct ITEM  m1[]=
{
                                /* Highlight  Char      Pos     */
        0, "Time Plot",         /*      Q              0       */
        0, "Frequency Plot",    /*       C             0       */
        0, "Convert File",      /*      R              0       */
        0, "Quit",              /*      R              0       */
        0, NULL
};
ClearBox(2,16,8,33,6,1);
ret1 = Menu(2, 15, m1, 0);
switch(ret1)
{
        case ESC: return ESC;
        case U_RT: ClearTextWindow( 2,15,12,35,_TBLUE ); return U_RT;
        case U_LT: ClearTextWindow( 2,15,12,35,_TBLUE ); return U_LT;
        case 0 :  Flag = SubMenuTimePlot(); return Flag;
        case 1 :  Flag = SubMenuFreqPlot(); break
        case 2 :  Flag = SubMenuCodeData();break;
        case 3 :  Flag = 0; break;
}
}                                       /* End of function DisplayMenu.                */


int  Assemblemenu( void )
{
int ret1,flag,r1,r2,c1,c2,lns,att;
struct ITEM  m1[]=
{
                                /* Highlight  Char      Pos     */
        0, "Concatenate ",      /*      Q              0       */
        0, "Smooth",            /*      C              0       */
        0, "Time Warp",         /*      C              0       */
```

```
        0, "Mainmenu",          /*      R           0       */
        0, "Quit",              /*      R           0       */
        0, NULL
};
ClearBox(2,60,9,75,7,1);
ret1 = Menu(2, 59, m1, 0);
switch(ret1)
{
        case ESC: return ESC;
        case U_RT: ClearTextWindow( 2,59,12,80,_TBLUE ); return U_RT;
        case U_LT: ClearTextWindow( 2,59,12,80,_TBLUE ); return U_LT;
        case 0 : break;
        case 1 : break;
        case 2 : break;
        case 3 : flag = 0; break;
        case 4 : flag = 0; break;
}
}                                       /* End of function AssembleMenu.          */


int Librarymenu( void )
{
int ret1,flag,r1,r2,c1,c2,lns,att;
struct ITEM  m1[]=
{
                        /* Highlight  Char      Pos     */
        0, "Open ",     /*      O           0       */
        0, "Close",     /*      C           0       */
        0, "Save",      /*      S           0       */
        0, "Mainmenu ", /*      M           0       */
        0, "Quit",      /*      Q           0       */
        0, NULL
};
ClearBox(2,47,9,59,7,1);
ret1 = Menu(2, 46, m1, 0);
switch(ret1)
{
        case ESC: return ESC;
        case U_RT: ClearTextWindow( 2,46,12,62,_TBLUE ); return U_RT;
        case U_LT: ClearTextWindow( 2,46,12,62,_TBLUE ); return U_LT;
        case 0 : break;
        case 1 : break;
```

```
        case 2 : break;
        case 3 : flag = 0; break;
        case 4 : flag = 0; break;
}
}                                                   /* End of function LibraryMenu.          */

int  Splicemenu( void )
{
int ret1,flag,r1,r2,c1,c2,lns,att;
struct ITEM  m1[]=
{
                        /* Highlight  Char      Pos     */
        0, "Time Plot",  /*      T             0       */
        0, "Freq Plot",  /*      F             0       */
        0, "Mainmenu",   /*      M             0       */
        0, "Quit",       /*      Q             0       */
        0, NULL
};
ClearBox(2,31,8,43,6,1);
ret1 = Menu(2, 30, m1, 0);
switch(ret1)
{
        case U_RT: ClearTextWindow( 2,30,12,45,_TBLUE ); return U_RT;
        case U_LT: ClearTextWindow( 2,30,12,45,_TBLUE ); return U_LT;
        case 0 : SubMenuTimePlot(); break;
        case 1 : SubMenuFreqPlot(); break;
        case 2 : break;
        case 3 : flag = 0; break;
}
}                                                   /* End of function SpliceMenu.           */


int  SubMenuFreqPlot( void )
{
return 0;
}                                                   /* End of function SubMenuFreqPlot.      */

void  submenuinit( void )
{
int   ret1,ret2,Seconds,r1,c1,r2,c2,lns,att,divisor;
struct MENU mnuAtrib =
```

```
{
        _TBLACK, _TBLACK, _TWHITE, _TBRIGHTWHITE, _TBRIGHTWHITE,
        _TWHITE, _TWHITE, _TBLACK, _TWHITE, _TBLACK,
        TRUE,
        '/', 'ø', 'Ÿ', '¿', '≥', 'ƒ'
};
struct ITEM  m2[4]=
{       1, " 115.2k,8,N,1",
        2, "  57.6k,8,N,1",
        3, "   MainMenu  ",
        0,NULL
};
r1 = 0; c1 = 0; r2 = 5; c2 = 30; lns = 5; att = 0;
ClearBox(2,16,7,31,5,1);
ret2 = Menu(2,14, m2, 0);
switch(ret2)
{
        case 0 : init1152(); _clearscreen( _GCLEARSCREEN ); break;
        case 1 : init1152(); ClearBox(r1,c1,r2,c2,lns,att); break;
        case 2 : ClearBox(r1,c1,r2,c2,lns,att); break;
}
}                                                   /* End of function SubMenuInit.                    */

void submenurecord( void )
{
        int   ret1,ret2,Seconds,r1,c1,r2,c2,lns,att;
        struct ITEM  m2[6]=
        {               1, " 2 Seconds  ",
                        2, "  4 Seconds  ",
                        3, "  16 Seconds  ",
                        4, "   32 Seconds ",
                        5, " Main Menu   ",
                        0,NULL
        };
        r1 = 0; c1 = 0; r2 = 24; c2 = 80; lns = 0; att = 0;
        ClearBox(3,16,10,32,7,1);
        ret2 = Menu(3, 14, m2, 0);
        switch(ret2)
        {
                        case 0 : Seconds = 2;
                                SendRecordCommand(Seconds);
```

```
                    record(Seconds);ClearBox(r1,c1,r2,c2,lns,att);
                    break;
            case 1 : Seconds = 4;
                    SendRecordCommand(Seconds);
                    record(Seconds);ClearBox(r1,c1,r2,c2,lns,att);
                    break;
            case 2 : Seconds  = 16;
                    SendRecordCommand(Seconds);
                    record(Seconds);ClearBox(r1,c1,r2,c2,lns,att);
                    break;
            case 3 : Seconds  = 32;
                    SendRecordCommand(Seconds);
                    record(Seconds);ClearBox(r1,c1,r2,c2,lns,att);
                    break;
            case 4 : ClearBox(r1,c1,r2,c2,lns,att);
                    break;
        }
}                                   /* End of function SubMenuRecord.          */

int  submenuplayback( void )
{
int  ret2, seconds, Flag, AccessCode = Read_Write;
int FileHandle, Blocks = 0x80;
unsigned long int           PlayBytes;
unsigned int                PlayBytesLow;
unsigned int                PlayBytesHigh;
unsigned int                FileOffsetLow = 0;
unsigned int                FileOffsetHigh = 0;
unsigned int                Time = 0;
char                        FileName[50];
char                        *Addr_FileName = &FileName[0];
struct ITEM  m2[6]=
{
        1, " 2 Seconds  ",
        1, " 4 Seconds  ",
        2, "  8 Seconds  ",
        3, "16 Seconds  ",
        4, "32 Seconds  ",
        0,NULL
};
ClearBox(4,16,11,32,7,1);
ret2 = Menu(4, 14, m2, 0);
```

```
Flag = GetUserInputFileName( Addr_FileName );
switch( Flag )
{
        case ESC:
        case U_LT:
        case U_RT: return ESC;
        default: break;
}
Time = 2*(int)( Power( (double)2, (double)ret2 ) );
PlayBytes = (unsigned long int)(4096) * (unsigned long int)(Time);
LongToShort( &PlayBytes, &PlayBytesLow, &PlayBytesHigh );
FileHandle = OpenFile( Addr_FileName, (unsigned char)AccessCode );
Playback(FileHandle,FileOffsetLow,FileOffsetHigh,PlayBytesLow,PlayBytesHigh,Blocks );
closefile( FileHandle );
ClearBox(0,0,24,80,0,0);
}                                                   /* End of function SubMenuPlayBack.          */

int GetFileSelection( struct ITEM FileNames[],  int ret1 )
{
extern unsigned long int FileSize[10];
extern unsigned int SizeOfFile[20];
int i = 0, flag = 1, j;
char FileInfo[43], FileName[50];
char *Addr_FileName = &FileName[0];
switch( ret1 )
{
        case ESC: return ESC;
        case 0: strcpy( Addr_FileName, "c:\\qc2\\ken\\cfiles\\*.pcm" );
                break;
        case 1: strcpy( Addr_FileName, "c:\\qc2\\ken\\cfiles\\*.adm" );
                break;
}
_asm            {
FileInfoAddr:   mov     ah,1ah                  /*Function number                   */
                mov     dx,WORD PTR Addr_FileInfo   /*File Info Buffer              */
                int     21h                     /*transfer to MS - DOS              */
FindFirstFile:  mov     ah,4eh                  /*Function number                   */
                mov     cx,0                    /*Normal attribute                  */
                mov     dx,WORD PTR Addr_FileName   /*Address of file name          */
                int      21h                    /*Transfer to MS - DOS              */
                jnc     okay
```

```
nomatch:        mov     flag,0
okay:           nop
                mov     cx,4
                xor     di,di
                mov     si,26
                xor     ax,ax
getfilesize:    mov     al,FileInfo[si]
                mov     BYTE PTR FileSize[di],al
                mov     al,BYTE PTR FileSize[di]
                mov     BYTE PTR SizeOfFile[di],al
                inc     si
                inc     di
                loop    getfilesize
                }

while( flag != 0 )
{
        FileNames[i].iHilite = 0;
        strcpy( FileNames[i].achItem, &FileInfo[30] );
        i = i + 1;
        _asm    {
FileInfoBuff:   mov     ah,1ah                  /*Function number               */
                mov     dx,WORD PTR Addr_FileInfo /*File Info Buffer            */
                int     21h                     /*transfer to MS – DOS          */
FindNextFile:   mov     ah,4fh                  /*Function number               */
                mov     cx,0                    /*Normal attribute              */
                mov     dx,WORD PTR Addr_FileName /*Address of file name        */
                int     21h                     /*Transfer to MS – DOS          */
                jnc     ok
nomorematch:    mov     flag,0
ok:             nop
                mov     si,26
                mov     cx,4
getfilesize1:   mov     al,FileInfo[si]
                mov     BYTE PTR FileSize[di],al
                mov     BYTE PTR SizeOfFile[di],al
                inc     si
                inc     di
                loop    getfilesize1
                }
}
```

```
FileNames[ i ].iHilite = 0;
FileNames[ i ].achItem[ 0 ] = NULL;
}                                                   /* End of function GetFileSelection.          */

/*      GetUserInputFileName writes the file name of the user selected file to location stringptr.    */
int  GetUserInputFileName( char *stringptr )
{
extern unsigned long int filesize;
extern unsigned long int FileSize[10];
extern unsigned int FileSizeLow;
extern unsigned int FileSizeHigh;
extern unsigned int SizeOfFile[20];
unsigned int Low;
int ret1, Flag, FileType;
struct ITEM  m1[10];
struct ITEM  FileTypeSelection[]=
{
                                /* Highlight  Char       Pos     */
        0, "PCM Format",        /*      P               0       */
        0, "ADPCM Format",      /*      A               0       */
        0, "Other",             /*      O               0       */
        0, NULL
};
char Header[30];
strcpy( Header, "Choose a file:");
strcpy( stringptr, "c:\\qc2\\ken\\cfiles\\" );
Box(10,10,10,60);
_settextposition( 11,32 );
_settextcolor( _TBLACK );
_outtext( Header );
FileType = Menu(12,15,FileTypeSelection,0);
switch( FileType )
{
        case ESC:
        case U_LT:
        case U_RT: return ESC;
        default: break;
}
Flag = GetFileSelection( m1, FileType );
switch( Flag )
{
        case ESC:
```

```
        case U_LT:
        case U_RT: return ESC;
        default: break;
}
ret1 = Menu(12, 15, m1, 0);
switch(ret1)
{
        case ESC:
        case U_RT:
        case U_LT: return ESC;
        default: strcpy( (stringptr +18), m1[ret1].achItem );
                filesize = FileSize[ret1];
                FileSizeLow = SizeOfFile[2*ret1];
                FileSizeHigh = SizeOfFile[2*ret1 + 1];
                break;
}
}                                       /* End of function GetUserInputFileName.      */


void ShowMainMenu( void )
{
        int i, r = 1, c = 2;
        _displaycursor( _GCURSOROFF );
        _setbkcolor( (long)_TRED);
        _clearscreen( _GCLEARSCREEN );
        for( i = 0; i < 5; i++ )
        {
                Itemize1( r, c, FALSE, aItem[i], 10);
                c = c + 15;
        }
}                                       /*End of function ShowMainMenu.              */


/*                              Choose from main menu function.                      */
int ChooseFromMenu( void )
{
        int i, r = 1, c = 2, CurCol = 2, PrevCol, Flag = 1;
        int cItem, cchItem = 2;         /* Counts of items and chars per item         */
        int iPrev, iCur = 0;            /* Indexes – temporary and previous.          */
        int acchItem[MAXITEM];          /* Array of counts of character in items.     */
        char *pchT;                     /* Temporary character pointer.               */
```

```
char achHilite[36];                    /* Array for highlight characters.                    */
unsigned uKey;                         /* Unsigned key code.                                 */
long bgColor;                          /* Screen color, position, and cursor.                */
short fgColor;
struct rccoord rc;
unsigned fCursor;
_setbkcolor( (long)_TBLUE );
_clearscreen( _GCLEARSCREEN );
fCursor = _displaycursor( _GCURSOROFF );
for( i = 0; i < 5; i++ )
{
        Itemize(r,c,FALSE,aItem[i],10);
        c = c + 15;
}
Itemize( r, CurCol,TRUE, aItem[iCur], 4 );
/* Count items, find longest, and put count of each in array. Also,
* put the highlighted character from each in a string.*/
for( cItem = 0; aItem[cItem].achItem[0]; cItem++ )
{
        acchItem[cItem] = strlen( aItem[cItem].achItem );
        cchItem=(acchItem[cItem]>cchItem) ? acchItem[cItem] :cchItem;
        i = aItem[cItem].iHilite;
        achHilite[cItem] = aItem[cItem].achItem[i];
}
cchItem += 2;
achHilite[cItem] = 0;          /* Null-terminate and lowercase string   */
strlwr( achHilite );
while( Flag = !0 && Flag != ESC )
{
        /* Wait until a uKey is pressed, then evaluate it.        */
        uKey = GetKey( WAIT );
        switch( uKey )
        {
                case ESC:   return ESC;
                case U_RT:                              /* Right key     */
                        Flag = -1;
                        iPrev = iCur;
                        if ( iCur < 4 )
                                iCur = iCur + 1;
                        else
                                iCur = 0;
```

```
                    PrevCol = CurCol;
                    if( CurCol < 48)
                            CurCol = CurCol + 15;
                    else
                            CurCol = 2;
            break;
            case U_LT:                          /* left key        */
                    Flag = -1;
                    iPrev = iCur;
                    if ( iCur > 0 )
                            iCur = iCur - 1;
                    else
                            iCur = 4;
                    PrevCol = CurCol;
                    if( CurCol > 2 )
                            CurCol = CurCol - 15;
                    else
                            CurCol = 62;
            break;
            default:
            if( uKey > 256 )  /* Ignore unknown key       */
                    continue;
/* If in highlight string, evaluate and fall through                    */
            pchT = strchr( achHilite, (char)tolower( uKey ) );
            if( pchT != NULL )  /* If in highlight string,              */
            {
                    iPrev = iCur;
                    iCur = pchT - achHilite;
                    PrevCol = CurCol;
                    CurCol = 2 + iCur*15;
            }
            else
                    continue;           /* Ignore unknown ASCII key   */
            Itemize( r, CurCol,TRUE, aItem[iCur],
            cchItem - acchItem[iCur] );
            Itemize(r,PrevCol,FALSE,aItem[iPrev],
            cchItem - acchItem[iPrev]);
            case ENTER:
                    _setbkcolor( bgColor );
                    _settextcolor( fgColor );
                    _settextposition( rc.row, rc.col );
                    _displaycursor( fCursor );
```

```
/* Flag=0 means Quit, Flag=ESC means go to Main menu*/
        while( Flag!=0 && Flag!=-1 && Flag!=ESC )
        {
        switch( iCur )
        {
                case 0: Flag = Mainmenu();
                switch( Flag )
                {
                        case ESC:       return ESC;
                        case U_RT:
                                iPrev = iCur;
                                iCur = 1;
                                PrevCol = CurCol;
                                CurCol = 17;
                                break;
                        case U_LT:
                                iPrev = iCur;
                                iCur = 4;
                                PrevCol = CurCol;
                                CurCol = 62;
                                break;
                        case 0: return 0;  /* Quit */
                }
                break;
                case 1: Flag = Displaymenu();
                        switch( Flag )
                        {
                        case QUIT:  return QUIT;
                        case ESC:   return ESC;
                        case U_RT:  iPrev = iCur;
                                iCur = 2;
                                PrevCol = CurCol;
                                CurCol = 32;
                        break;
                        case U_LT:  iPrev = iCur;
                                iCur = 0;
                                PrevCol = CurCol;
                                CurCol = 2;
                        break;
                        }
                break;
```

```
case 2: Flag = Splicemenu();
        switch( Flag )
        {
        case ESC:   return ESC;
        case U_LT:  iPrev = iCur;
                    iCur = 1;
                    PrevCol = CurCol;
                    CurCol = 17;
        break;
        case U_RT:  iPrev = iCur;
                    iCur = 3;
                    PrevCol = CurCol;
                    CurCol = 47;
        break;
        }
break;
case 3: Flag = Librarymenu();
        switch( Flag )
        {
        case ESC:   return ESC;
        case U_LT:  iPrev = iCur;
                    iCur = 2;
                    PrevCol = CurCol;
                    CurCol = 32;
        break;
        case U_RT:  iPrev = iCur;
                    iCur = 4;
                    PrevCol = CurCol;
                    CurCol = 62;
        break;
        }
break;
case 4: Flag = Assemblemenu();
        switch( Flag )
        {
        case ESC:   return ESC;
        case U_LT:  iPrev = iCur;
                    iCur = 3;
                    PrevCol = CurCol;
                    CurCol = 47;
        break;
```

```
                                        case U_RT:  iPrev = iCur;
                                                    iCur = 0;
                                                    PrevCol = CurCol;
                                                    CurCol = 2;
                                        break;
                                        }
                                break;
                                }
                                /* Redisplay current and previous. */
                                Itemize(r,CurCol,TRUE,aItem[iCur],
                                cchItem – acchItem[iCur]);
                                Itemize(r,PrevCol,FALSE,aItem[iPrev],
                                cchItem–acchItem[iPrev]);
                        }
                }
                /* Redisplay current and previous. */
                Itemize(r,CurCol,TRUE, aItem[iCur], cchItem – acchItem[iCur]);
                Itemize(r,PrevCol,FALSE,aItem[iPrev],cchItem–acchItem[iPrev]);
        }
}                                       /* End of function ChooseFromMainMenu.          */
```

/*Function Menu – Puts menu on screen and reads menu input from keyboard. When a highlighted hot key or ENTER is pressed, returns the index of the selected menu item.

Params: row and col – If "fCentered" attribute of "mnuAtrib" is true,center row and column of menu; otherwise top left of menu

aItem – array of structure containing the text of each item and the index of the highlighted hot key

iCur – index of the current selection—pass 0 for first item, or maintain a static value

Return: The index of the selected item

Uses: mnuAtrib */

```
int Menu( int row, int col, struct ITEM aItem[], int iCur )
{
        int cItem, cchItem = 2; /* Counts of items and chars per item     */
        int i, iPrev;           /* Indexes – temporary and previous       */
        int acchItem[MAXITEM]; /* Array of counts of character in items   */
        char *pchT;             /* Temporary character pointer            */
        char achHilite[36];   /* Array for highlight characters          */
        unsigned uKey;        /* Unsigned key code                       */
        long bgColor;         /* Screen color, position, and cursor      */
        short fgColor;
        struct rccoord rc;
        unsigned fCursor;
```

```
/* Save screen information. */
        fCursor = _displaycursor( _GCURSOROFF );
        bgColor = _getbkcolor();
        fgColor = _gettextcolor();
        rc = _gettextposition();
/* Count items, find longest, and put count of each in array. Also, put the highlighted character from each
in a string.                                                                                    */
for( cItem = 0; aItem[cItem].achItem[0]; cItem++ )
        {
                acchItem[cItem] = strlen( aItem[cItem].achItem );
                cchItem = (acchItem[cItem] > cchItem) ? acchItem[cItem] : cchItem;
                i = aItem[cItem].iHilite;
                achHilite[cItem] = aItem[cItem].achItem[i];
        }
        cchItem += 2;
         achHilite[cItem] = 0;          /* Null–terminate and lowercase string  */
        strlwr( achHilite );
/* Adjust if centered, and draw menu box. */
        if( mnuAtrib.fCentered )
        {
                row -= cItem / 2;
                col -= cchItem / 2;
        }
        Box( row++, col++, cItem, cchItem );
/* Put items on menu. */
        for( i = 0; i < cItem; i++ )
        {
                if( i == iCur )
                        Itemize( row + i, col, TRUE, aItem[i], cchItem – acchItem[i] );
                else
                        Itemize( row + i, col, FALSE, aItem[i], cchItem – acchItem[i] );
        }
        while( TRUE )
        {                                       /* Wait until a uKey is pressed, then evaluate it. */
                uKey = GetKey( WAIT );
                switch( uKey )
                {
                        case ESC:   return ESC;
                        case U_RT:
                        case U_LT:
                        return uKey;
```

```
                    case U_UP:                          /* Up key       */
                    iPrev = iCur;
                    iCur = (iCur > 0) ? (—iCur % cItem) : cItem – 1;
                    break;
                    case U_DN:                          /* Down key     */
                    iPrev = iCur;
                    iCur = (iCur < cItem) ? (++iCur % cItem) : 0;
                    break;
                    default:
                    if( uKey > 256 )            /* Ignore unknown function key  */
                    continue;
                    pchT = strchr( achHilite, (char)tolower( uKey ) );
                    if( pchT != NULL )           /* If in highlight string,    */
                    iCur = pchT – achHilite;/*   evaluate and fall through  */
                    else
                    continue;                   /* Ignore unknown ASCII key       */
                    case ENTER:
                    _setbkcolor( bgColor );
                    _settextcolor( fgColor );
                    _settextposition( rc.row, rc.col );
                    _displaycursor( fCursor );
                    return iCur;
                }
/* Redisplay current and previous. */
        Itemize( row + iCur, col,
        TRUE, aItem[iCur], cchItem – acchItem[iCur] );
        Itemize( row + iPrev, col,
        FALSE, aItem[iPrev], cchItem – acchItem[iPrev] );
        }
}

/* ClearTextWindow– Draw menu box, filling interior with blanks of the border color.
 Params: row and col – upper left of box rowLast and colLast – height and width
 Return: None
 Uses:   mnuAtrib                                                              */

void  ClearTextWindow(int row, int col, int rowLast, int colLast, int color )
{
        int i;
        unsigned char far *ptr_blanks;
        unsigned char blankstring[80];
```

```
        ptr_blanks = &blankstring[0];
        for( i = 0; i <= (colLast – col – 1); i++ )
        {
                blankstring[ i ] = ' ';
        }
        blankstring[ i + 1 ] = NULL;
        /* Set color and position. */
        _settextposition( row, col );
        _settextcolor( (short) color );
        _setbkcolor( (long) color );
        for( i = 0; i <= (rowLast – row); ++i )
        {
                _settextposition( row + i, col );
                _outtext( ptr_blanks );
        }
}
/* Box – Draw menu box, filling interior with blanks of the border color.
Params: row and col – upper left of box
rowLast and colLast – height and width
Return: None
Uses:   mnuAtrib                                                        */

void Box( int row, int col, int rowLast, int colLast )
{
        int i;
        char achT[MAXITEM + 2];  /* Temporary array of characters */
        /* Set color and position. */
        _settextposition( row, col );
        _settextcolor( mnuAtrib.fgBorder );
        _setbkcolor( mnuAtrib.bgBorder );
        /* Draw box top. */
        achT[0] = mnuAtrib.chNW;
        memset( achT + 1, mnuAtrib.chEW, colLast );
        achT[colLast + 1] = mnuAtrib.chNE;
        achT[colLast + 2] = 0;
        _outtext( achT );
        /* Draw box sides and center. */
        achT[0] = mnuAtrib.chNS;
        memset( achT + 1, ' ', colLast );
        achT[colLast + 1] = mnuAtrib.chNS;
        achT[colLast + 2] = 0;
```

```
        for( i = 1; i <= rowLast; ++i )
        {
                _settextposition( row + i, col );
                _outtext( achT );
        }
        /* Draw box bottom. */
        _settextposition( row + rowLast + 1, col );
        achT[0] = mnuAtrib.chSW;
        memset( achT + 1, mnuAtrib.chEW, colLast );
        achT[colLast + 1] = mnuAtrib.chSE;
        achT[colLast + 2] = 0;
        _outtext( achT );
}

/* Itemize – Display one selection (item) of a menu. This function is normally only used internally by
Menu.
Params: row and col – top left of menu
fCur – flag set if item is current selection
itm – structure containing item text and index of highlight
cBlank – count of blanks to fill
Return: none
Uses:   mnuAtrib                                                                    */

void Itemize( int row, int col, int fCur, struct ITEM itm, int cBlank )
{
        int i;
        char achT[MAXITEM];             /* Temporary array of characters */
        /* Set text position and color. */
        _settextposition( row, col );
        if( fCur )
        {
                _settextcolor( mnuAtrib.fgSelect );
                _setbkcolor( mnuAtrib.bgSelect );
        }
        else
        {
                _settextcolor( mnuAtrib.fgNormal );
                _setbkcolor( mnuAtrib.bgNormal );
        }
        /* Display item and fill blanks. */
        strcat( strcpy( achT, " " ), itm.achItem );
        _outtext( achT );
```

```
        memset( achT, ' ', cBlank— );
        achT[cBlank] = 0;
        _outtext( achT );
        /* Set position and color of highlight character, then display it. */
        i = itm.iHilite;
        _settextposition( row, col + i + 1 );
        if( fCur )
        {
                _settextcolor( mnuAtrib.fgSelHilite );
                _setbkcolor( mnuAtrib.bgSelHilite );
        }
        else
        {
                _settextcolor( mnuAtrib.fgNormHilite );
                _setbkcolor( mnuAtrib.bgNormHilite );
        }
        _outchar( itm.achItem[i] );
}

void Itemize1( int row, int col, int fCur, struct ITEM itm, int cBlank )
{
        char achT[MAXITEM];             /* Temporary array of characters */
        /* Set text position and color. */
        _settextposition( row, col );
        _settextcolor( mnuAtrib.fgNormal );
        _setbkcolor( mnuAtrib.bgNormal );
        /* Display item and fill blanks. */
        strcat( strcpy( achT, " " ), itm.achItem );
        _outtext( achT );
        memset( achT, ' ', cBlank— );
        achT[cBlank] = 0;
        _outtext( achT );
}

/* GetKey – Gets a key from the keyboard. This routine distinguishes between ASCII keys and function or
control keys with different shift states. It also accepts a flag to return immediately if no key i available.
Params: fWait – Code to indicate how to handle keyboard buffer:
NO_WAIT         Return 0 if no key in buffer, else return key
WAIT Return first key if available, else wait for key
CLEAR_WAIT  Throw away any key in buffer and wait for new key
Return: One of the following:
```

```
        Keytype                 High Byte       Low Byte
        -------                                 ---------       ---------
No key available (only with NO_WAIT)            0               0
ASCII value     0       ASCII code
Unshifted function or keypad                    1       scan code
Shifted function or keypad              2       scan code
CTRL function or keypad                         3       scan code
ALT function or keypad                          4       scan code
Note:           getkey cannot return codes for keys not recognized by BIOS
int 16, such as the CTRL-UP or the 5 key on the numeric keypad.                 */

unsigned GetKey( int fWait )
{
        unsigned uKey, uShift;
        /* If CLEAR_WAIT, drain the keyboard buffer. */
        if( fWait == CLEAR_WAIT )
                while( _bios_keybrd( _KEYBRD_READY ) )
                        _bios_keybrd( _KEYBRD_READ );
        /* If NO_WAIT, return 0 if there is no key ready. */
        if( !fWait && !_bios_keybrd( _KEYBRD_READY ) )
                return FALSE;
        /* Get key code. */
        uKey = _bios_keybrd( _KEYBRD_READ );
        /* If low byte is not zero, it's an ASCII key. Check scan code to see
         * if it's on the numeric keypad. If not, clear high byte and return.
         */
        if( uKey & 0x00ff )
                if( (uKey >> 8) < 69 )
                        return( uKey & 0x00ff );
        /* For function keys and numeric keypad, put scan code in low byte
         * and shift state codes in high byte.
         */
        uKey >>= 8;
        uShift = _bios_keybrd( _KEYBRD_SHIFTSTATUS ) & 0x000f;
        switch( uShift )
        {
                case 0:
                        return( 0x0100 | uKey ); /* None (1)       */
                case 1:
                case 2:
                case 3:
```

```
                return( 0x0200 | uKey ); /* Shift (2)   */
        case 4:
                return( 0x0300 | uKey ); /* Control (3) */
        case 8:
                return( 0x0400 | uKey ); /* Alt (4)  */
    }
}
unsigned GetControlKey( int fWait )
{
    unsigned uKey, uShift;
    /* Get key Flag. */
    uKey = GetKeyboardControlFlag();
    switch( uKey )
    {
        case 0:
                return( 0x0100 | uKey ); /* None (1)        */
        case 1:
        case 2:
        case 3:
                return( 0x0200 | uKey ); /* Shift (2)   */
        case 4:
                return( 0x0300 | uKey ); /* Control (3) */
        case 8:
                return( 0x0400 | uKey ); /* Alt (4)  */
    }
}

unsigned int GetKeyboardControlFlag( void )
{
    unsigned uKey;
    _asm    {
    readflag:       mov     ah,02h
                    int     16h
                    test  al,08h
                    jz      readflag
                    mov     uKey,400h
            }
    return uKey;
}
```

/* _outchar – Display a character. This is the character equivalent of _outtext. It is affected by _settextposition, _settextcolor, and _setbkcolor. It should not be used in loops. Build strings and then

```
_outtext to show multiple characters.
Params: out – character to be displayed
Return: none                                                    */

void _outchar( char out )
{
static char achT[2] = " ";     /* Temporary array of characters */
achT[0] = out;
_outtext( achT );
}

void end_program( void )
{
        _displaycursor( _GCURSORON );
        _clearscreen( _GCLEARSCREEN );
        _setvideomode( _DEFAULTMODE );
}
```

A.2.2 Record

```
#include <string.h>
#include <fprotyp7.h>
#define NULL 0

void record( int Seconds )
{
int             FileHandle, bufseg, TwoSecCount;
char            FileName[50];

strcpy( FileName, "c:\\qc2\\ken\\cfiles\\test.adm" );   /*Name of file to be recorded.          */
FileHandle = CreateAndOpenFile( FileName, 2 );          /*Create file.                           */
bufseg = AllocateMemory( 512 );                         /*Allocate RAM for speech data.         */
TwoSecCount = Seconds/2;

_asm            {
                push    es              /*Remember C's extra segment value.                     */
                mov     ax,bufseg       /*es contains the starting address of the segment       */
                mov     es,ax           /*where speech data will be buffered.                   */

notrdy:         mov     dx,03feh        /*Test whether Slave is requesting to send.             */
                in      al,dx           /*Slave is req. to send if CTS=1, since Slave's         */
                and     al,10h          /*RTS=0 when Slave is requesting to send.               */
                je      notrdy          /*b4=0 means not rts: b4=1 means request to send.       */

again: xor      di,di
                mov     cx,8192         /*Two seconds of speech.                                */

more: mov       al,03h                  /*Slave: it is clear to send.                           */
                mov     dx,03fch
                out     dx,al

notrdy:         mov     dx,03fdh        /*Test for reception of complete character.             */
                in      al,dx           /*                                                      */
                shr     al,1
                jnc     notrdy

                xor     al,al           /*Having received a character, Host now informs         */
                mov     dx,03fch        /*the Slave not to send.                                */
                out     dx,al           /*RTS=0 means not clear to send.                        */
```

```
                mov     dx,03f8h        /*Read the data byte from the serial port.      */
                in      al,dx

                mov     es:[di],al      /*Write the byte to RAM.                        */
                inc     di              /*Increment RAM pointer.                        */
                loop    more            /*Loop if 8192 bytes have not been recorded.    */

writef1:        mov     ah,40H          /*Write function number.                        */
                mov     bx,FileHandle   /*FileHandle is a number associated with a file.*/
                push    ds              /*Remember C's data segment.                    */
                mov     dx,bufseg       /*ds contains the segment where speech data     */
                mov     ds,dx           /*is buffered in RAM.                           */
                mov     dx,0            /*dx contains the  offset of 1st RAM byte.      */
                mov     cx,8192         /*Write 8K bytes to disk.                       */
                int     21h             /*DOS write RAM to file interrupt routine.      */
                jc      Error           /*c = 1 means error                             */
                pop     ds              /*Restore C's data segment.                     */
                dec     TwoSecCount     /*Check if done recording.                      */
                jz      rtn
                jmp     again
Error: pop      ds                      /*Restore C's data segment.                     */
                nop                     /*Place error code here.                        */
rtn:            pop     es              /*Restore C's extra segment.                    */
                }                       /*End asm routine.                              */
ReleaseMemory( bufseg );
closefile (FileHandle );
}                                       /*End Record function.                          */
```

```
void SendRecordCommand( int Seconds )
{
_asm {
                mov     al,02h          /*Slave: Host is requesting to send.                    */
                mov     dx,03fch        /*RTS = 1 means request to send.                       */
                out     dx,al

notrdy:         mov     dx,03feh        /*Test whether Slave is ready to receive.              */
                in      al,dx           /*Slave is ready to receive if CTS=1, since Slave's    */
                and     al,10h          /*RTS=0 when Slave is requesting to send.              */
                je      notrdy          /*b4=0 means not rts: b4=1 means request to send.      */

TranShftFull:   mov     dx,03fdh        /*Check whether transmit shift reg is empty.           */
                in      al,dx
                and     al,40h
                je      TranShftFull    /*b6=1 means transmit shift reg is empty.              */

transmit:       mov     dx,03f8h        /*Now transmit.                                        */
                mov     al,BYTE PTR Seconds     /*Type cast Seconds to byte.                   */
                out     dx,al           /*al contains the record command code.                 */

                mov     cx,01000h       /*Wait timer.                                          */
NotValid:       mov     dx,03feh        /*Wait for Slave's indication of a valid command       */
                in      al,dx           /*                                                     */
                and     al,10h
                je      Valid           /*b4=0 means valid command.                            */
                loop    NotValid
                jmp     notrdy          /*If time out, try sending command again.              */

Valid: mov      al,00h                  /*host sends acknowledge.                              */
                mov     dx,03fch        /* i.e., host not requesting to send.                  */
                out     dx,al
                }                       /*End asm routine.                                     */
}                                       /*End send record command function.                    */
```

A.2.3 Playback

```
#include <fprotyp6.h>
void Playback(      int                 FileHandle,
                    unsigned int        FileOffsetLow,
                    unsigned int        FileOffsetHigh,
                    unsigned int        LSWSelectedBytes,
                    unsigned int        MSWSelectedBytes,
                    int                 Blocks      )
{
unsigned int        SmallInnerLoop, InnerLoop, OuterLoop;
unsigned char       Data[8192];
unsigned char       *StartofData = &Data[0];

OuterLoop = MSWSelectedBytes*8 + LSWSelectedBytes/8192;
InnerLoop = 8192;
SmallInnerLoop = LSWSelectedBytes%8192;
SendPlaybackCommand( 170 );

_asm        {
            call    SetFilePointer      /*Set file pointer to starting location of the*/
                                        /*portion of file selected by the user.*/
            cmp     OuterLoop,0h        /*What is the size of the playback file?*/
            jz      fin                 /*Q.  0 ≤ Size of playback < 8192?*/

start:      call    ReadFile            /*Read 8192 bytes from file referred to by*/
                                        /*FileHandle, and dump these bytes to*/
                                        /*RAM starting at the location pointed to by*/
                                        /*StartofData.*/
            mov     cx,InnerLoop        /*cx = number of bytes to transmit*/
            mov     si,WORD PTR StartofData     /*si point to start of data.*/

here:       mov     al,02h              /*Slave: Host is now requesting to send.*/
            mov     dx,03fch            /*This enables the slave to begin its*/
            out     dx,al               /*playback routine.*/

notrdy0:    mov     dx,03feh            /*Test if Slave is ready to receive.*/
            in      al,dx               /*Slave is ready to receive if its RTS is 1.*/
            and     al,10h              /*B4=0 indicates not ready to receive.*/
            je      notrdy0             /*b4=1 means Slave is ready to receive.*/
```

```
tdrfull:    mov   dx,03fdh      /*Check if transmit shift register is empty.*/
            in    al,dx
            and   al,40h
            jz    tdrfull       /*b6=1 means shift register is empty.*/

notrdy1:    mov   dx,03feh      /*Test if Slave is ready to receive.*/
            in    al,dx         /*Slave is ready to receive if its RTS is 1.*/
            and   al,10h        /*B4=0 indicates not ready to receive.*/
            je    notrdy1       /*b4=1 means Slave is ready to receive.*/

            mov   dx,03f8h      /*Now the Host transmits.*/
            mov   al,[si]       /*Write byte pointed to by si to the UART.*/
            out   dx,al
            inc   si            /*Increment the source pointer.*/

contloop:   loop  notrdy0       /*Finished transmitting InnerLoop bytes?.*/
            dec   OuterLoop     /*If finished transmitting InnerLoop bytes,*/
                                /*decrement OuterLoop counter.*/
            jz    fin           /*Any more 8192 blocks to transmit?*/
            jmp   Start

fin:  cmp   SmallInnerLoop,0h
            jz    finished

            mov   dx,SmallInnerLoop
            mov   InnerLoop,dx
            mov   OuterLoop,1h
            xor   dx,dx
            mov   SmallInnerLoop,dx
            jmp   start

finished:   mov   al,00h        /*Slave: Host is not requesting to send.*/
            mov   dx,03fch      /*This ends the playback routine, since*/
            out   dx,al         /*Slave stops playback when it senses that*/
                                /*Host has reset its RTS to logic 0*/
            jmp   rtn
```

```
/*Start of procedures.*/

/*This subroutine (procedure) increments the file pointer. Note, for as long as a file is open, the operating
system, DOS, updates the file pointer each time the file is read from or written to. Therefore, this function,
Playback, does not need to increment the file pointer. This proceedure is included here to show how
incrementing the file pointer may be done.*/
IncFilePointer: mov     ax,InnerLoop            /*Increment the file pointer by Innerloop.*/
                add     FileOffsetLow,ax
                adc     FileOffsetHigh,0
                ret

/*This proceedure sets the file pointer to the starting location in the portion of the file the user had selected.
FileOffsetHigh and FileOffsetLow are the file pointers selected by the user.*/
SetFilePointer: mov     ax,4200h                /*ah = 42h is the function number.*/
                mov     bx,FileHandle           /*Filepointer referenced from start of file.*/
                mov     cx,FileOffsetHigh       /*Most significant half of offset */
                mov     dx,FileOffsetLow        /*Least significant half of offset*/
                int     21h                     /*Set file pointer interrupt routine.*/
                jc      error                   /*carry = 1 means error*/
                ret                             /*Return from proceedure.*/
error:          nop                             /*Place error code here.*/
                ret                             /*Return from proceedure.*/

/*This proceedure reads InnerLoop number of bytes from file referenced by FileHandle and dumps these
bytes to RAM starting at location pointed by StartofData.*/

ReadFile:       mov     ah,3fh                  /*Read function number.*/
                mov     bx,FileHandle           /*FileHandle references C:\qc2\ken\*.adm.*/
                mov     dx,WORD PTR StartofData /*dx=offset of 1st RAM byte.*/
                mov     cx,InnerLoop            /*read InnerLoop bytes from file.*/
                int     21h
                jc      Readerror               /*Carry = 1 means error.*/
                ret                             /*Return from proceedure.*/
ReadError:      nop                             /*Place error code here.*/
                ret                             /*Return from proceedure.*/
rtn:            nop

/*End of procedures.*/

                }                               /*End of asm routine.*/
}                                               /*End of function Playback.*/
```

```
void SendPlaybackCommand( int Blocks )
{
_asm {
                mov     al,02h          /*Slave: Host is requesting to send.                    */
                mov     dx,03fch        /*RTS = 1 means request to send.                       */
                out     dx,al

notrdy:         mov     dx,03feh        /*Test whether Slave is ready to receive.              */
                in      al,dx           /*Slave is ready to receive if CTS=1, since Slave's    */
                and     al,10h          /*RTS=0 when Slave is requesting to send.              */
                je      notrdy          /*b4=0 means not rts: b4=1 means request to send.      */

TranShftFull:   mov     dx,03fdh        /*Check whether transmit shift reg is empty.           */
                in      al,dx
                and     al,40h
                je      TranShftFull    /*b6=1 means transmit shift reg is empty.              */

transmit:       mov     dx,03f8h        /*Now transmit.                                        */
                mov     al,BYTE PTR Blocks      /*Type cast Seconds to byte.                   */
                out     dx,al           /*al contains the record command code.                 */
                mov     cx,01000h       /*Wait timer.                                          */
NotValid:       mov     dx,03feh        /*Wait for Slave's indication of a valid command       */
                in      al,dx           /*                                                     */
                and     al,10h
                je      Valid           /*b4=0 means valid command.                            */
                loop    NotValid
                jmp     notrdy          /*If time out, try sending command again.              */

Valid: mov      al,00h                  /*host sends acknowledge.                              */
                mov     dx,03fch        /* i.e., host not requesting to send.                  */
                out     dx,al
                }                       /*End asm routine.                                     */
}                                       /*End send playback command function.                  */
```

A.2.4 Time Plot

```
#include<def1.h>
#include<fprotyp1.h>
#include<string.h>
#include<stdio.h>
unsigned long int GetSizeOfFile( int FileHandleClipboard );
void Insert( int FileHandle1, unsigned int File1OffsetLow, unsigned int File1OffsetHigh, unsigned long int
InsertSize, int FileHandle2, unsigned int File2OffsetLow, unsigned int File2OffsetHigh ,int
FileHandleADPCMBak);

int SubMenuTimePlot( void )
{
extern unsigned long int   filesize;
extern unsigned int        FileSizeLow;
extern unsigned int        FileSizeHigh;
unsigned long int          FileOffsetPCM = 0;
unsigned long int          FileOffsetADPCM = 0;
unsigned long int          FileOffsetPointer1 = 0;
unsigned long int          FileOffsetPointer2 = 0;
unsigned long int          SelectedBytesADPCM;
unsigned long int          Size;
unsigned char              Data[600];
unsigned char              *StartofData = &Data[0];
unsigned int               FileOffsetPCMHigh = 0;
unsigned int               FileOffsetPCMLow = 0;
unsigned int               FileOffsetADPCMHigh = 0;
unsigned int               FileOffsetADPCMLow = 0;
unsigned int               SelectedBytesADPCMHigh = 0;
unsigned int               SelectedBytesADPCMLow = 0;
unsigned int               InnerLoop = 600;
unsigned int               OuterLoop = 1;
float                      x;
char                       FileNamePCM [100 ];
char                       FileNamePCMBak [100 ];
char                       FileNameADPCM [100 ];
char                       FileNameADPCMBak[ 100 ];
char                       FileNameClipboard[ 31 ];
char                       CopyMessage2[ 20 ];
char                       *AddrFileNamePCM = &FileNamePCM[ 0 ];
char                       *AddrFileNamePCMBak = &FileNamePCMBak[ 0 ];
```

```
char                    *AddrFileNameADPCM = &FileNameADPCM[ 0 ];
char                    *AddrFileNameADPCMBak = &FileNameADPCMBak[ 0 ];
char                    *Addr_FileNameClipboard = &FileNameClipboard[ 0 ];
char                    *Addr_CopyMessage2 = &CopyMessage2[ 0 ];
int                     FileHandlePCM;
int                     FileHandleADPCM;
int                     FileHandleADPCMBak;
int                     FileHandlePCMBak;
int                     Byte_Size = 600;
int                     i;
int                     j = 15;
int                     Flag;
int                     PointerFlag = 0;
int                     Blocks = 0x80;
int                     *OuterLoopPtr = &OuterLoop;
int                     FileHandleClipboard;
int                     MouseCoordinates[2];
int                     Pointer1 = -1;
int                     Pointer2 = -1;
int                     NumberHotBoxes = 10;
int HotBoxes[] = {      570,174,630,184,        10,174,70,184,          10,158,39,168,
                        601,158,630,168,        40,158,600,168,         20,0,620,129,
                        200,174,250,184,        260,174,310,184,        320,174,380,184,
                        100,174,130,184                                                 };

Flag = GetUserInputFileName( AddrFileNamePCM );
switch( Flag )
{
        case ESC:
        case U_RT:
        case U_LT:          return ESC;
        default:    Flag = 33;
        break;
}
/*strcpy automatically terminates a string with a NULL character.*/
strcpy(         Addr_FileNameClipboard, "c:\\qc2\\ken\\cfiles\\Clipbord.dat" );
strcpy(         Addr_CopyMessage2,"Bytes selected:  " );
strcpy(         AddrFileNamePCMBak,    AddrFileNamePCM         );
strcpy(         (AddrFileNamePCMBak + strlen(AddrFileNamePCMBak)-3),"PBK");
strcpy(         AddrFileNameADPCM,     AddrFileNamePCM         );
strcpy(         (AddrFileNameADPCM + strlen(AddrFileNameADPCM)-3),"adm");
```

```
strcpy(         AddrFileNameADPCMBak,       AddrFileNameADPCM );
strcpy(         (AddrFileNameADPCMBak + strlen(AddrFileNameADPCM)-3),"ADB");

FileHandleClipboard = CreateAndOpenFile( Addr_FileNameClipboard, (unsigned char)2 );
FileHandlePCM = OpenFile( AddrFileNamePCM, Read_Write );
FileHandleADPCM = OpenFile( AddrFileNameADPCM, Read_Write );
FileHandleADPCMBak = CreateAndOpenFile( AddrFileNameADPCMBak, Read_Write );
FileHandlePCMBak = CreateAndOpenFile( AddrFileNamePCMBak, Read_Write );
graphics_mode();
DrawOscilliscope();
DrawPlotmenuIcon();
ResetMouse();
PlotData(       FileHandlePCM,      FileOffsetPCMLow,   FileOffsetPCMHigh,
                Byte_Size,          StartofData,        &Data[0],
                FileOffsetPCM                                               );
ShowMouse();
while( Flag != QUIT )
{
Flag = GetMouseSelection( &HotBoxes[0], NumberHotBoxes, &MouseCoordinates[0] );

    switch( Flag )
    {

    case 1: Flag = QUIT;        /*User selected quit.                       */
    break;

    case 2:                     /*User selected playback.                   */
    if( PointerFlag == 2 )
    {
            if( FileOffsetPointer1 < FileOffsetPointer2 )
            {
                FileOffsetADPCM = FileOffsetPointer1/2;
                LongToShort(    &FileOffsetADPCM,   &FileOffsetADPCMLow,
                                &FileOffsetADPCMHigh                        );
                SelectedBytesADPCM = (FileOffsetPointer2-FileOffsetPointer1 )/2;
            }
            else
            {
                FileOffsetADPCM = FileOffsetPointer2/2;
                LongToShort(    &FileOffsetADPCM,   &FileOffsetADPCMLow,
                                &FileOffsetADPCMHigh );
```

```
                SelectedBytesADPCM = ( FileOffsetPointer1–FileOffsetPointer2 )/2;
        }
        LongToShort(    &SelectedBytesADPCM,  &SelectedBytesADPCMLow,
                &SelectedBytesADPCMHigh                                    );
        Playback(       FileHandleADPCM,              FileOffsetADPCMLow,
                        FileOffsetADPCMHigh,  SelectedBytesADPCMLow,
                        SelectedBytesADPCMHigh,       Blocks               );
}
else
{
        Playback(       FileHandleADPCM,              FileOffsetADPCMLow,
                        FileOffsetADPCMHigh,  InnerLoop*OuterLoop,
                        0,                            Blocks               );
}
        break;


case 3:                 /*User selected move plotting window left by 60 bytes.      */
        if( FileOffsetPCM > (unsigned long int)60 )
        {
                FileOffsetPCM = FileOffsetPCM – (unsigned long int)60;
                LongToShort(    &FileOffsetPCM,       &FileOffsetPCMLow,
                        &FileOffsetPCMHigh                                 );
        }
        else
        {
                FileOffsetPCM = (unsigned long int)0;
                FileOffsetPCMLow = (unsigned int)0;
        }
        ClearGraphicsScreen( 1,20,1,620,128 );
        PlotData(       FileHandlePCM, FileOffsetPCMLow,
                        FileOffsetPCMHigh,    Byte_Size,      StartofData,
                        &Data[0],             FileOffsetPCM                );
        RememberMarkers(        Pointer1,     Pointer2,       FileOffsetPCM,
                                FileOffsetPointer1,           FileOffsetPointer2   );
        break;

case 4:                 /*User selected move plotting window right by 60 points.    */
        if( FileOffsetPCM < (filesize – (unsigned long int)660) )
        {
                FileOffsetPCM = FileOffsetPCM + (unsigned long int)60;
                LongToShort(    &FileOffsetPCM,       &FileOffsetPCMLow,
```

```
                                &FileOffsetPCMHigh                              );
        }
        else
        {
                FileOffsetPCM = (filesize – (unsigned long int)600);
                LongToShort(    &FileOffsetPCM,         &FileOffsetPCMLow,
                                &FileOffsetPCMHigh                              );
        }
        ClearGraphicsScreen( 1,20,1,620,128 );
        PlotData(       FileHandlePCM, FileOffsetPCMLow,
                        FileOffsetPCMHigh,      Byte_Size,      StartofData,
                        &Data[0],               FileOffsetPCM                   );
        RememberMarkers(        Pointer1,       Pointer2,       FileOffsetPCM,
                                FileOffsetPointer1,             FileOffsetPointer2      );
        break;

case 5:                 /*Mouse click in scroll bar rectangle:  course movement.        */
        HideMouse();
        ClearGraphicsScreen( 1, 41, 159, 599, 167 );
        DrawRectangle( 3, 40, 159, MouseCoordinates[0], 167);
        FileOffsetPCM = (long)(((MouseCoordinates[0]–40.0)/560)*(filesize–600));
        LongToShort(    &FileOffsetPCM,         &FileOffsetPCMLow,
                        &FileOffsetPCMHigh                              );
        FileOffsetADPCM = FileOffsetPCM/2;
        LongToShort(    &FileOffsetADPCM,       &FileOffsetADPCMLow,
                &FileOffsetADPCMHigh                                    );
        ClearGraphicsScreen( 1,20,1,620,128 );
        PlotData(       FileHandlePCM, FileOffsetPCMLow,
                        FileOffsetPCMHigh,      Byte_Size,      StartofData,
                        &Data[0],               FileOffsetPCM                   );
        RememberMarkers(        Pointer1,       Pointer2,       FileOffsetPCM,
                                FileOffsetPointer1,             FileOffsetPointer2      );
        ShowMouse();
        break;

case 6:  /*User selected position pointers in file window.                        */
HideMouse();

if(     MouseCoordinates[0] == (Pointer1 + 20)  ||
        MouseCoordinates[0] == (Pointer2 + 20)          )
{       /*User selected erase pointer in file window.                           */
```

```
            if( MouseCoordinates[0] == Pointer1 + 20 )
                    Pointer1 = -1;
            else
                    Pointer2 = -1;
            DrawLine(       MouseCoordinates[0],        1,      MouseCoordinates[0],
                            128,                        0                               );
            PlotData(       FileHandlePCM, FileOffsetPCMLow,
                            FileOffsetPCMHigh,  Byte_Size,      StartofData,
                            &Data[0],           FileOffsetPCM                   );
            PointerFlag—;
            ClearTextLine( 19, 1, 45 );
}
else
{           /*User selected to position pointer in file window.                   */
            if( PointerFlag < 2 )
            {
                    if( Pointer1 == -1 )
                    {
                            Pointer1 = MouseCoordinates[0] – 20;
                            FileOffsetPointer1 = (long)(Pointer1) + FileOffsetPCM;
                    }
                    else
                    {
                            Pointer2 = MouseCoordinates[0] – 20;
                            FileOffsetPointer2 = (long)(Pointer2) + FileOffsetPCM;
                    }
            DrawLine(       MouseCoordinates[0],        1,      MouseCoordinates[0],
                            128,                        7                               );
            PointerFlag++;
            }
}
if( PointerFlag == 2 )
{           /*User attempting to position more than two pointers.                 */
            ClearTextLine( 19, 1, 45 );
            PrintGrahicsText( 19, 2, Addr_CopyMessage2 );
            printf("%lu", AbsLong( FileOffsetPointer1, FileOffsetPointer2 ));
}
ShowMouse();
break;

case 7:             /*User selects copy speech to clipboard file.               */
```

```
if( PointerFlag == 2 )
{               /*Enable to copy only if both pointers are positioned.          */
        if( FileOffsetPointer1 > FileOffsetPointer2 )
        {
                SelectedBytesADPCM=( FileOffsetPointer1-FileOffsetPointer2 )/2;
                LongToShort(    &FileOffsetPointer2,    &FileOffsetADPCMLow,
                                &FileOffsetADPCMHigh                    );
        }
        else
        {
                SelectedBytesADPCM=( FileOffsetPointer2-FileOffsetPointer1 )/2;
                LongToShort(    &FileOffsetPointer1,    &FileOffsetADPCMLow,
                        &FileOffsetADPCMHigh                    );
        }
        SelectedBytesADPCM=AbsLong(FileOffsetPointer1, FileOffsetPointer2)/2;
        LongToShort(    &SelectedBytesADPCM,  &SelectedBytesADPCMLow,
                &SelectedBytesADPCMHigh                         );
        Copy(   FileHandleADPCM,        FileOffsetADPCMLow,
                FileOffsetADPCMHigh,  FileHandleClipboard,      0,      0,
                SelectedBytesADPCMLow,      SelectedBytesADPCMHigh          );
        ClearTextLine( 19, 1, 45 );
}
break;

case 8:         /*Cut selection from file                                       */
        Flag = QUIT;
break;

case 9:         /*Paste selected speech from clipboard file into current file.  */
        if( PointerFlag == 1 )          /*Paste if one pointer is positioned.   */
        {
                if( Pointer1 != -1 )
                {
                        FileOffsetADPCM = FileOffsetPointer1/2;
                }
                else
                {
                        FileOffsetADPCM = FileOffsetPointer2/2;
                }
                Size = GetSizeOfFile( FileHandleClipboard );
                LongToShort(    &FileOffsetADPCM,    &FileOffsetADPCMLow,
                        &FileOffsetADPCMHigh                    );
```

```
                    Insert(  FileHandleClipboard,    (unsigned int)0,    (unsigned int)0,
                             Size,                   FileHandleADPCM,
                             FileOffsetADPCMLow,     FileOffsetADPCMHigh,
                             FileHandleADPCMBak                                        );
                }
        break;

        case 10:        /*User selected all speech data                                */
                FileOffsetPointer1 = 0;
                FileOffsetPointer2 = filesize;
                Pointer1 = 0;
                Pointer2 = 600;
                RememberMarkers(    Pointer1,       Pointer2,       FileOffsetPCM,
                                    FileOffsetPointer1,             FileOffsetPointer2);
                PointerFlag = 2;
                ClearTextLine( 19, 1, 45 );
                PrintGrahicsText( 19, 2, Addr_CopyMessage2 );
                printf("%lu", AbsLong( FileOffsetPointer1, FileOffsetPointer2 ));
        break;
        default: Flag = QUIT;
        }
}
        closefile( FileHandleClipboard );
        closefile( FileHandlePCM );
        closefile( FileHandleADPCM );
        closefile( FileHandlePCMBak );
        closefile( FileHandleADPCMBak );
        end_program();
        return ESC;
}
/*Function RememberMarkers draws a previously selected marker if that marker is pointing to a file
position that is in the current view window.  Figure A.3 explains the main idea of this function.
                                                                                */
```

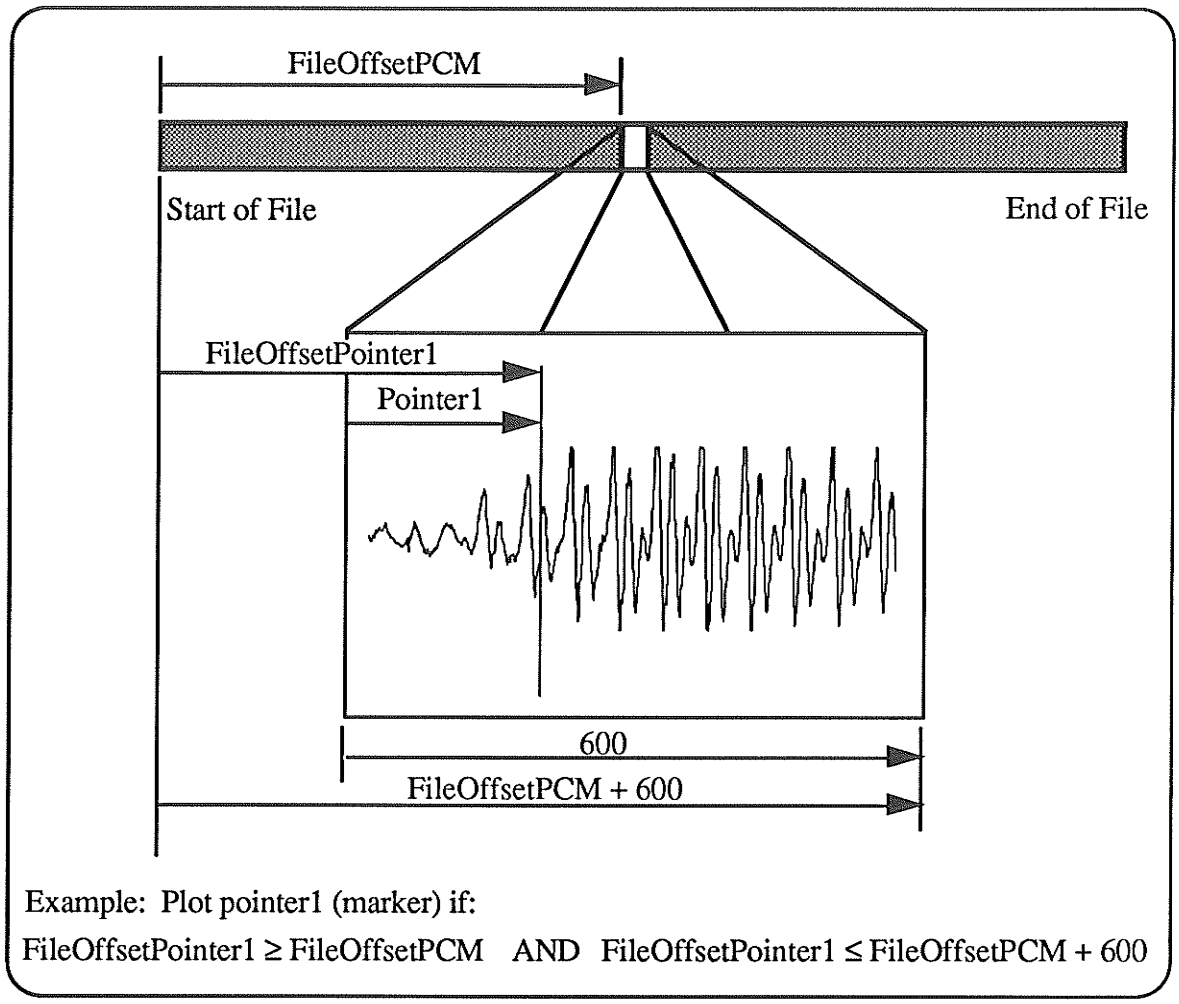


Fig. A.3 Plot marker calculation.

```
void RememberMarkers(     int Pointer1, int Pointer2,
                          unsigned long int    FileOffsetPCM,
                          unsigned long int    FileOffsetPointer1,
                          unsigned long int    FileOffsetPointer2          )
{
if(                                   Pointer1 != -1 &
FileOffsetPointer1 ≥ FileOffsetPCM & FileOffsetPointer1 ≤ (FileOffsetPCM + 600)          )
    {
        Pointer1 = (short)( (FileOffsetPointer1 – FileOffsetPCM) );
        DrawLine(      Pointer1 + 20, 1, Pointer1 + 20, 128, 7                    );
    }
if(                                   Pointer2 != -1 &
FileOffsetPointer2 ≥ FileOffsetPCM & FileOffsetPointer2 ≤ (FileOffsetPCM + 600)    )
    {
        Pointer2 = (short)( (FileOffsetPointer2 – FileOffsetPCM) );
```

```
                DrawLine(       Pointer2 + 20,  1,  Pointer2 + 20,  128,  7                 );
        }
}
```

A.2.5 File I/O

```
/*Function Prototypes*/

int                 OpenFile(               char *addr_fname1,char AccessCode                       );
int                 CreateAndOpenFile(      char *AddrFName, char AccessCode                        );
int                 SetFilePointer(         int FileHandle,     int LowOffset,      int HighOffset  );
int                 DumpFiletoRAM(          int FileHandle,     int HighBytes,      int LowBytes,
                                            int RAMAddress                                          );
int                 SaveRAMtoFile(          int FileHandle,     int bytes,          int RAMAddress  );
void                closefile(              int FileHandle                                          );
unsigned long int   GetSizeOfFile(          int FileHandleClipboard                                 );

void                    Insert(             int                 FileHandle1,
                                            unsigned int        File1OffsetLow,
                                            unsigned int        File1OffsetHigh,
                                            unsigned long int   InsertSize,
                                            int                 FileHandle2,
                                            unsigned int        File2OffsetLow,
                                            unsigned int        File2OffsetHigh,
                                            int                 FileHandle2         );

void                    Copy(               int                 FileHandle1,
                                            unsigned int        File1OffsetLow,
                                            unsigned int        File1OffsetHigh,
                                            int                 FileHandle2,
                                            unsigned int        File2OffsetLow,
                                            unsigned int        File2OffsetHigh,
                                            unsigned int        BytesLow,
                                            unsigned int        BytesHigh           );
```

/*Function Open File

OpenFile function opens a file that has been previously created. It accepts two parameters:

1) the address of the ASCIIZ file name (ASCIIZ is the ASCII name of the file followed by a 0).

2) the Access Code.

A FileHandle is returned which can be used for subsequent access to the file. The function calls BIOS int 21h with registers initialized as follows:

ah	3dh	function code
al	AccessCode	file access attributes
dx	AddrFName	offset of ASCIIZ path name

Access Code:

al	**Bits(0–2)** type of access	
	000	read
	001	write
	010	read and write
	Bit(3)	reserved
	Bits(4–6) sharing mode	
	000	compatibility
	001	deny all
	010	deny write
	011	deny read
	100	deny none
	Bit(7)	inheritance flag
	0	child process inherits handle
	1	child does not inherits handle

BIOS returns status as follows:

if function successful

CF	clear
ax	filehandle

if function unsuccessful

CF	set
ax	error code

Error Codes:

ax	02h	file not found
ax	03h	path not found

ax	04h	no handles available
ax	05h	access denied
ax	0ch	invalid access code

Notes: • After opening the file, the file pointer is reset to the beginning of the file.

```
int OpenFile( char *AddrFName, char AccessCode )
{
int FileHandle;
_asm {
                mov     ah,3dh
                mov     al,AccessCode
                mov     dx,AddrFName
                int     21h
                jc      Error
                mov     FileHandle,ax
                jmp     Return
Error:          nop                     /*Place error code here         */
Return:         nop
                }                       /*End asm routine               /*
return FileHandle;
}
```

/*Function CreateAndOpenFile

CreateAndOpenFile function either creates a new file or opens and truncates an existing file to zero length. The function accepts two parameters:

1) the address of the ASCIIZ file name (ASCIIZ is the ASCII name of the file followed by a 0) and

2) the Access Code.

A FileHandle is returned which can be used for subsequent access to the file. The function calls BIOS int 21h function 3ch (create file) with registers initialized as follows:

ah	3ch	function code
cx	Bit(s)	file attribute
	0	normal
	1	hidden
	2	system
	3	volume label
	4	reserved(0)
	5	archive
	6–15	reserved(0)
dx		offset of ASCIIZ path name

BIOS returns status as follows:

if function successful

CF	clear
ax	filehandle

if function unsuccessful

CF	set
ax	error code

Error Codes:

ax	03h	path not found
ax	04h	no handles available
ax	05h	access denied

Notes: Access denied may indicate that there is no room for a directory entry or an existing file

is read only and can't be opened for output.

The function then calls BIOS int 21h function 3dh (open file) as discussed above.*/

```
int CreateAndOpenFile( char *AddrFName, char AccessCode )
{
int filehandle;
_asm            {
                mov     ah,3ch          /*create file with normal attribute     */
                xor     cx,cx
                mov     dx,AddrFName
                int     21h
                jc      Error
                mov     ah,3dh          /*now open the file                     */
                mov     al,AccessCode
                mov     dx,AddrFName
                int     21h
                jc      Error
                mov     filehandle,ax
                jmp     Return
Error:  nop                             /*Place error code here                 */
Return:         nop
                }                       /*End asm routine                       */
return filehandle;
}                                       /*End function                          */
```

/*Function Close File

Function closefile, BIOS int 21h function 3eh, flushes all internal buffers associated with the file to disk, closes the file, and releases the handle for reuse. If the file was modified the time and date stamp are updated. The function accepts one parameter, FileHandle.

The function calls BIOS int 21h with registers initialized as follows:

ah	3eh	function code
bx	FileHandle	file handle

•BIOS returns status as follows:

if function successful
CF clear
if function unsuccessful
CF set
ax error code

Error Code:

ax	06h	invalid handle

*/

```
void closefile( int FileHandle )
{
_asm {
        mov     ah,3eh
        mov     bx,FileHandle
        int     21h
        jc      Error
Error:  nop                     /*Place error code here         */
        }                       /*End asm routine               */
}                               /*End function                  */
```

/*Function Set FIle Pointer

Function SetFilePointer, BIOS int 21h function 42h, sets the file pointer relative to either the start of file, the end of file, or the current position. The function calls BIOS int 21h with registers initialized as

follows:

	ah	42h	function code
	al	method code:	relative method
		00h	absolute from start of file
		01h	signed offset from current file position
		02h	signed offset from end of file
	bx	FileHandle	file handle
	cx	LowOffset	most significant 16 bit offset
	dx	HighOffset	least significant 16 bit offset

BIOS returns status as follows:

if function successful

CF	clear	
dx	most significant 16 bit offset from start of file	
ax	least significant 16 bit offset from start of file	

if function unsuccessful

CF	set	
ax	error code	

Error codes:

ax	01h	invalid relative method
	06h	invalid handle

Notes: This function uses a long integer (HighOffset concatenated with the LowOffset) to set the file pointer. The next byte read or written to the file will be at the new file pointer dx:ax relative from the start of the file. */

```
int SetFilePointer( int FileHandle, int LowOffset, int HighOffset )
{
_asm {
        mov     ax,4200h        /*ah = 42h is the function number       */
        mov     bx,FileHandle   /*al = 0 ptr offset  start of file       */
        mov     cx,HighOffset   /*Most significant half of offset       */
        mov     dx,LowOffset    /*Least significant half of offset      */
        int     21h
        jnc     Return          /*carry = 1 means error                 */
        jmp     Error
Error:  nop                     /*place error code here                 */
Return:
        }                       /*End asm routine                       */
}                               /*End Function                          */
```

/*Function Dump A File to RAM

Function DumpFiletoRAM, BIOS int 21h function 3fh, transfers data from a file to RAM. The function accepts 3 parameters:

1) the FileHandle of the file of interest,

2) the number of bytes to transfer, and

3) the first address (RAMAddress) of RAM where the data will be dumped.

The data from the file is retrieved starting from the current file pointer position. The function calls BIOS int 21h with registers initialized as follows:

ah	3fh	function code
bx	FileHandle	file handle
cx	bytes	number of bytes to read
dx	RAMAddress	Address of first RAM location

BIOS returns status as follows:

if function successful

CF	clear
ax	number of bytes transferred

if function unsuccessful

CF	set	
ax	error code	

Error Codes:

ax	05h	access denied
	06h	invalid handle

Notes: If the CF = 0 but ax = 0, then the file pointer was at the end of file. */

```
int DumpFiletoRAM(   int FileHandle, int HighBytes, int LowBytes,
                     int RAMAddress )
{
_asm {
            mov     ah,3fh          /*read function number                  */
            mov     bx,FileHandle
            mov     dx,RAMAddress   /*offset of the first RAM location      */
            mov     cx,LowBytes     /*read bytes from file FileHandle       */
            int     21h
            jnc     Return          /*1 means error: ax has error code      */
Error:  nop                         /*place error code here                 */
Return:
            }                       /*End asm routine                       */
}                                   /*End Function                          */
```

/*Function Save RAM Area to Disk

Function SaveRAMtoFile, BIOS int 21h function 40h, transfers data from RAM to disk(file). The function accepts 3 parameters:

1) the FileHandle of the file of interest,

2) the number of bytes to transfer, and

3) the first address (RAMAddress) of RAM where the data is located.

The data from RAM is written to the file starting from the current file pointer position. The function calls BIOS int 21h with registers initialized as follows:

ah	40h	function code
bx	FileHandle	file handle
cx	bytes	number of bytes to read
dx	RAMAddress	Address of first RAM location

BIOS returns status as follows:

if function successful

CF	clear
ax	number of bytes transferred

if function unsuccessful

CF	set
ax	error code

Error Codes:

ax	05h	access denied
	06h	invalid handle

Notes: If CF = 0 but ax < cx, then the remaining data, ax – cx, could not written because of insufficient space on disk.

```
int SaveRAMtoFile( int FileHandle, int bytes, int RAMAddress )
{
_asm {
                mov     ah,40h          /*write function number                 */
                mov     bx,FileHandle
                mov     dx,RAMAddress   /*dx is the 1st RAM address             */
                mov     cx,bytes        /*write bytes of data to disk           */
                int     21h
                jnc     Return          /*C=1 means error                       */
Error:  nop                             /*place error code here                 */
Return:         nop
                }                       /*End asm routine                       */
}                                       /*End Function                          */
```

```
/*Function Copy copies BytesHigh:BytesLow bytes starting at location File1OffsetHigh:FIle1OffsetLow
from file FileHandle1 and writes these bytes to FileHandle2 starting at location
File2OffsetHigh:FIle2OffsetLow.                                                    */

BytesHigh:BytesLow                  32 bit unsigned size of bytes to copy.
FileHandle1                         Handle which references the source file.
File1OffsetHigh:FIle1OffsetLow      32 bit offset of source file.
FileHandle2                         Handle which references the destination file.
File2OffsetHigh:FIle2OffsetLow      32 bit offset of destination file.
OuterLoop                           Number of 8192 (8K) blocks to be copied.
InnerLoop                           8K bytes to be copied.
SmallInnerLoop                      Remainder upon division of 8K.
void  Copy(     int                 FileHandle1,
                unsigned  int       File1OffsetLow,
                unsigned  int       File1OffsetHigh,
                int                 FileHandle2,
                unsigned  int       File2OffsetLow,
                unsigned  int       File2OffsetHigh,
                unsigned  int       BytesLow,
                unsigned  int       BytesHigh          )
{
unsigned int        SmallInnerLoop;
unsigned int        InnerLoop;
unsigned int        OuterLoop;
unsigned char       Data[8192];
unsigned char       *StartofData = &Data[0];
OuterLoop = BytesHigh*8 + BytesLow/8192;
SmallInnerLoop = BytesLow%8192;
InnerLoop = 8192;

_asm                {
                    call        SetPointer
                    cmp         OuterLoop,0
                    jz           fin
Start:              call        ReadFile
                    call        WriteFile
                    dec         OuterLoop
                    jnz         Start
fin:                cmp         SmallInnerLoop,0
                    jz           rtn
                    mov         dx,SmallInnerLoop
```

```
                mov     InnerLoop,dx
                mov     OuterLoop,1h
                mov     SmallInnerLoop,0
                jmp     start

/*Proceedure SetPointer sets the file pointer to location File1OffsetHigh:File1OffsetLow with respect to the
start of file referenced by FileHandle1, which is the source file.                                          */
SetPointer:     mov     ax,4200h        /*ah = 42h is the function number                                   */
                mov     bx,FileHandle1  /*al = 0 ptr offset  start of file                                   */
                mov     cx,File1OffsetHigh  /*Most significant half of offset                               */
                mov     dx,File1OffsetLow   /*Least significant half of offset                              */
                int     21h
                jnc     Return          /*carry = 1 means error                                             */
Error:          ret                     /*place error code here                                             */
Return:         ret

/*Proceedure WriteFile writes InnerLoop bytes located at RAM address StartofData to disk referred to by
FileHandle2, which is the destination file.*/
WriteFile:      mov     ah,40h          /*Write function number                                             */
                mov     bx,FileHandle2
                mov     dx,WORD PTR StartofData     /*dx is the 1st RAM address                             */
                mov     cx,InnerLoop    /*Write InnerLoop bytes of data to disk                             */
                int     21h
                jc      WriteError      /*C=1 means error.                                                  */
                ret                     /*Return to calling program.                                        */
WriteError:     ret                     /*Place error code here.                                            */

/*Proceedure ReadFile reads Innerloop bytes from file referenced by FileHandle1 to RAM starting at
location StartofData.                                                                                       */

ReadFile:       mov     ah,3fh          /*Read function number                                              */
                mov     bx, FileHandle1 /*FileHandle1 references source file.                               */
                mov     dx,WORD PTR StartofData     /*Offset of first RAM location.                         */
                mov     cx, InnerLoop   /*Read InnerLoop bytes from file.                                   */
                int     21h
                jc      ReadError   /*1 means error: ax has error code.                                     */
                ret                     /*Return to calling program.                                        */
ReadError:      ret                     /*Place error code here.                                            */
                }                       /*End asm routine                                                   */
}                                       /*End Function Copy.                                                */
```

```
/*Function LongToShort converts a 32 bit unsigned long integer to two 16 bit short integers. FileOffset =
FileOffsetHigh:FileOffsetLow.                                                    */

void  LongToShort(      unsigned  int  long          *FileOffset,
                        unsigned  int                *FileOffsetLow,
                        unsigned  int                *FileOffsetHigh          )
{
_asm            {
                xor     si,si
                xor     di,di
                mov     si,WORD PTR FileOffset
                mov     di,WORD PTR FileOffsetLow
                mov     ax,[si]
                mov     [di],ax
                mov     di,WORD PTR FileOffsetHigh
                mov     ax,[si+2]
                mov     [di],ax
                }
}

/*Function GetSizeOfFile returns a 32 bit size of file FIlehandle.  This function calls DOS interrupt 21
function 42h.  This DOS routine moves the file pointer by cx:dx times relative to (specified by register al)
the start of the file, the current file pointer position, or the end of the file.  If the routine is successful, the
new file pointer position is returned in registers dx:ax relative to the starting of the file.  By specifying
moving the file pointer relative to the end of the file by 0 times, then dx:ax actually returns the size of the
file.                                                                            */
unsigned  long  int  GetSizeOfFile( int  FileHandle )
{
unsigned long int  SizeofFile = 0;
unsigned long int  *SizeFile = &SizeofFile;
_asm            {
                mov     di,WORD PTR SizeFile
                mov     ax,4202h        /*ah = 42h is the function number          */
                mov     bx,FileHandle   /*al=02 specifies moving relative to EOF.   */
                mov     cx,0            /*Most significant half of offset=0.       */
                mov     dx,0            /*Least significant half of offset=0.      */
                int     21h
                jnc      rtn            /*Carry = 1 means error.                   */
                jmp     errorcode
errorcode:      nop                     /*Place error code here.                   */
rtn:            mov     [di],ax         /*Load least significant 16 bit size first.*/
```

```
                inc     di
                inc     di
                mov     [di],dx         /*Most significant 16 bit size.          */
                }                       /*End asm routine.                       */
return SizeofFile;
}                                       /*End function GetSizeofFile.            */
```

/*Function Insert copies 32 bit InsertSize bytes from file FileHandle1 starting from File1OffsetHigh:FIle1OffsetLow from file FileHandle1 and inserts these bytes to FileHandle2Bak at location File2OffsetHigh:FIle2OffsetLow. The size of file FileHandle2Bak increases by Insertsize bytes.

BytesHigh:BytesLow	32 bit unsigned size of bytes to copy.
FileHandle1	Handle which references the source file.
File1OffsetHigh:FIle1OffsetLow	32 bit offset of source file.
FileHandle2	Handle which references the destination file.
FileHandle2Bak	Backup of file referred to by FIleHandle2.
File2OffsetHigh:FIle2OffsetLow	32 bit offset of destination file. */

```
void Insert(    int                     FileHandle1,
                unsigned int            File1OffsetLow,
                unsigned int            File1OffsetHigh,
                unsigned long int       InsertSize,
                int                     FileHandle2,
                unsigned int            File2OffsetLow,
                unsigned int            File2OffsetHigh,
                int                     FileHandle2Bak    )
{
extern unsigned long int    filesize;
unsigned long int   File2Offset;
unsigned long int   File3Offset;
unsigned int        InsertSizeLow, InsertSizeHigh,;
unsigned int        File3OffsetLow, File3OffsetHigh;

Copy( FileHandle2, (unsigned int)0, (unsigned int)0, FileHandle2Bak, (unsigned int)0, (unsigned int)0,
File2OffsetLow, File2OffsetHigh );

LongToShort( &InsertSize, &InsertSizeLow, &InsertSizeHigh );

Copy( FileHandle1, (unsigned int)0, (unsigned int)0, FileHandle2Bak, File2OffsetLow+1,
File2OffsetHigh, InsertSizeLow, InsertSizeHigh );

File2Offset= ( unsigned long int )File2OffsetLow + ( unsigned long int )File2OffsetHigh*65535;
```

```
File3Offset = File2Offset + InsertSize;
LongToShort(      &File3Offset,          &File3OffsetLow,                &File3OffsetHigh          );
InsertSize = filesize/((unsigned long int)2) – InsertSize;
LongToShort( &InsertSize, &InsertSizeLow, &InsertSizeHigh );
Copy( FileHandle2, File2OffsetLow, File2OffsetHigh, FileHandle3, File3OffsetLow, File3OffsetHigh,
InsertSizeLow, InsertSizeHigh   );
}                                                  /*End function Insert.                                  */
```

A.2.6 Graphics

```
#include <graph.h>
#include <stdio.h>
#include <process.h>
struct videoconfig myscreen;
void    PlotData( int FileHandle, unsigned int FileOffset_Low, unsigned int FileOffset_High,
int Byte_Size, unsigned char *StartofData, unsigned char *Data, unsigned long int FileOffset );
void    PrintGrahicsText(int Row, int Col,char *Addr_Text );
void    DrawRectangle( int FillFlag, int x1, int y1, int x2, int y2 );
void    ClearTextLine( int Row, int Col, int Spaces );
void    SetFilePointer( int FileHandle, int LowOffset, int HighOffset );
void    FiletoRAM( int FileHandle, int HighBytes, int LowBytes, int RAMAddress );
void    DrawOscilliscope( void ;
void    DrawPlotmenuIcon( void );
void    PrintNumberPlayPages( unsigned int OuterLoop );
void    DrawLine( int x1, int y1, int x2, int y2, int Color );

void    graphics_mode( void )
{
_getvideoconfig( &myscreen );
        switch( myscreen.adapter )
        {
                case _CGA:
                        _setvideomode( _HRESBW );
                        break;
                case _OCGA:
                        _setvideomode( _ORESCOLOR );
                        break;
                case _EGA:
                case _OEGA:
                        if( myscreen.monitor == _MONO )
                                _setvideomode( _ERESNOCOLOR );
                        else
                                _setvideomode( _ERESCOLOR );
                        break;
                case _VGA:
                case _OVGA:
                case _MCGA:
                        _setvideomode( _VRES2COLOR );
                        break;
```

```
                case _HGC:
                        _setvideomode( _HERCMONO );
                        break;
                default:
                        printf( "This program requires a CGA, EGA, VGA, or Hercules card\n" );
                        exit( 0 );
        }
        _getvideoconfig( &myscreen );
}

void DrawLine( int x1, int y1, int x2, int y2, int Color )
{
int PrevColor = _getcolor();
        _setcolor( Color );
        _moveto( x1, y1 );
        _lineto( x2, y2 );
        _setcolor( PrevColor );
}

void PrintNumberPlayPages( unsigned int OuterLoop )
{
                _settextposition( 23,14 );
                _outtext("  ");
                _settextposition( 23,14 );
                printf( "%d", OuterLoop );
}

void DrawPlotmenuIcon( void )
{
        unsigned char RightArrow[] = { 0x1a, 0 };
        unsigned char LeftArrow[] = { 0x1b, 0 };

        _rectangle(_GBORDER,10,160,620,170);

        _rectangle(_GBORDER,70,174,100,184);
        _settextposition( 23,11 );
        _outtext( LeftArrow );

        _rectangle(_GBORDER,130,174,160,184);
        _settextposition( 23,19 );
        _outtext( RightArrow );
```

```
        _rectangle(_GBORDER,100,174,130,184);
        _settextposition( 23,14 );
        _outtext( "1" );

        _rectangle(_GBORDER,10,174,70,184);
        _settextposition( 23,4 );
        _outtext( "Play" );

        _rectangle(_GBORDER,200,174,250,184);
        _settextposition( 23,27 );
        _outtext( "Copy" );

        _rectangle(_GBORDER,260,174,310,184);
        _settextposition( 23,35 );
        _outtext( "Cut" );

        _rectangle(_GBORDER, 320,174,380,184);
        _settextposition( 23,42 );
        _outtext( "Paste" );

        _rectangle(_GBORDER, 560,174,620,184);
        _settextposition( 23,73 );
        _outtext( "Main" );
}

void  DrawOscilliscope(void)
{
int i, j;
_rectangle( _GBORDER, 15,0,630,129 );
        for( j =2; j <= 122; j = j + 20 )
        {
                for ( i=12; i <=18; i++ )
                {
                        _setpixel( i,j );
                        _setpixel( i + 615,j );
                }
        }
        _settextposition( 18,33 );
        _outtext( "Time (seconds)" );
        _settextposition( 1,0 );
```

```
        printf("5\n \n \nA\nm\np\nl\ni\nt\nu\nd\ne\n \n \n \n0");
}


void  PlotData(       int  FileHandle,              unsigned  int  FileOffset_Low,
                      unsigned  int  FileOffset_High,     int  Byte_Size,
                      unsigned  char  *StartofData,       unsigned  char  *Data,
                      unsigned  long  int  FileOffset                                )
{
        int j = 20, i;
        SetFilePointer( FileHandle, FileOffset_Low, FileOffset_High );
        DumpFiletoRAM( FileHandle, 0, Byte_Size, (int)StartofData );
        _setviewport( 0,0,620,199 );
        for( i = 0; i <= 599; i++ )
        {
                j = j + 1;
                _setpixel( j, (( unsigned int )(*( Data + i )))/2);
        }
                _settextposition( 18,2 );
                _outtext( "                    " );
                _settextposition( 18,2 );
                printf( "%.3f", (double)(FileOffset)/8192.0 );
                _settextposition( 18,73 );
                _outtext( "                    " );
                _settextposition( 18,73 );
                printf( "%.3f", (double)(FileOffset)/8192.0 + .073 );
}

void  ClearTextLine( int Row, int Col ,int Spaces )
{
int i;
        for( i = 0; i < Spaces; i++ )
        {
                _settextposition( Row, Col + i );
                _outtext( " " );
        }
}

void  ClearGraphicsScreen( int ViewPortFlag,int x1,int y1,int x2,int y2 )
{
                _setviewport( x1,y1,x2,y2 );
                _clearscreen( ViewPortFlag ); /*ViewPortFlag=1=_GVIEWPORT*/
```

```
}

void DrawRectangle( int FillFlag,int x1,int y1,int x2,int y2)
{
      _setviewport( 0,0,620,199 );
      _rectangle(FillFlag,x1,y1,x2,y2); /*fillFlag=3=fill interior*/
}
void PrintGrahicsText( int Row, int Col, char *Addr_Text )
{
      _settextposition( Row, Col );
      _outtext( Addr_Text );
}
```

A.2.7 Mouse

This section lists the functions associated with the mouse. Figure M1 shows a typical sequence of events in a mouse polling routine.

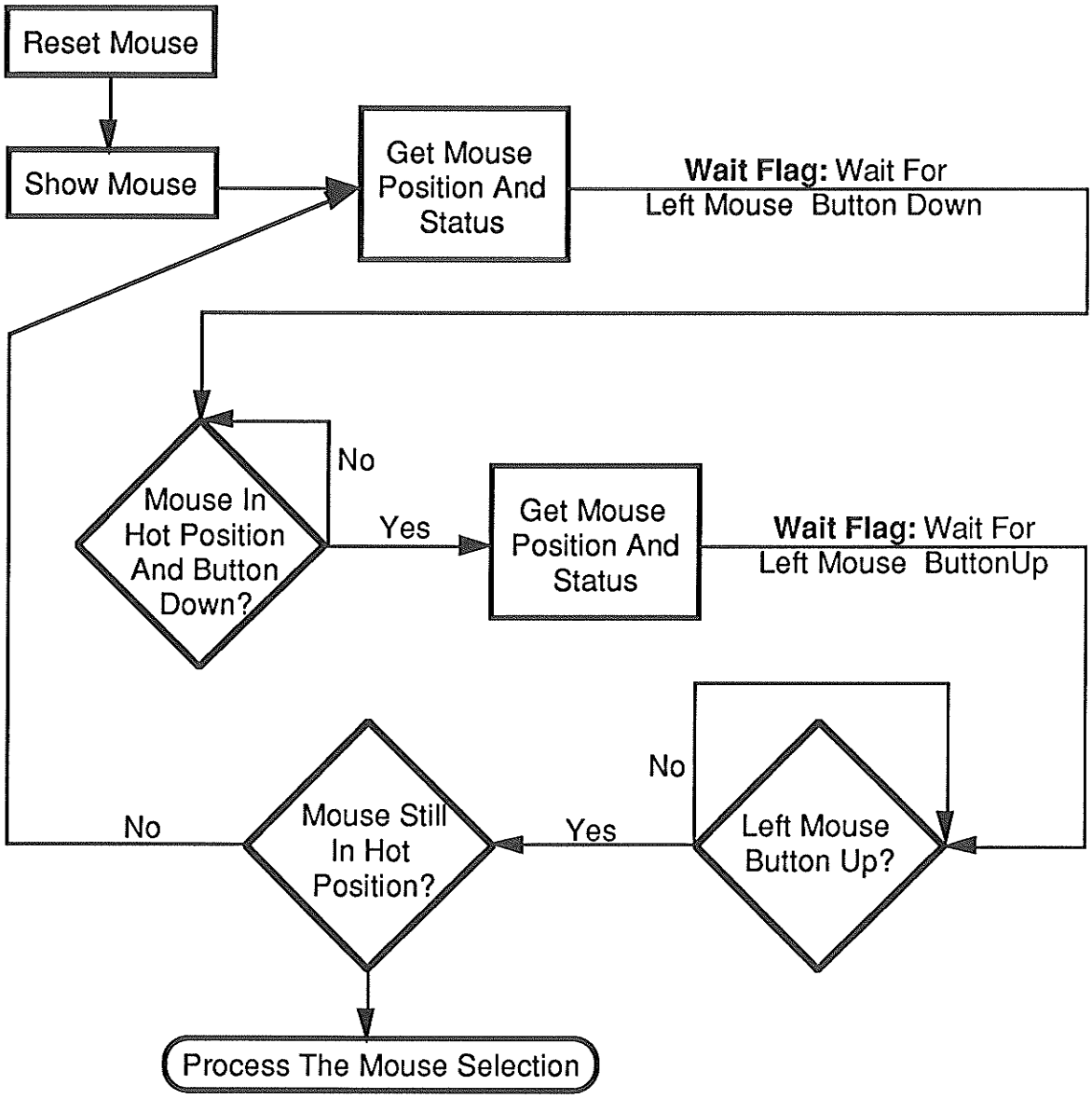


Figure A.1: Mouse polling routine. 1) Reset the Mouse, 2) Show the Mouse, 3) Wait for left mouse button down, then check whether mouse is in hot position, 4) If in hot position, wait for mouse button up, then check again whether mouse is in same hot position, and 4) If in same hot position, then process mouse selection, else start the polling routine at 1).

```
int ResetMouse( void )
{
_asm {
        xor     ax,ax
        int     33h
        }                               /*end asm routine.                  */
}                                       /*end function ResetMouse.          */
```

```
int ShowMouse( void )
{
_asm {
        mov     ax,1
        int     33h
        }                               /*end asm routine.                  */
}                                       /*end function    Show mouse.       */
```

/*Function Get Mouse Position And Button Status

The GetMousePositionAndButtonStatus function returns the address of an array containing the mouse position in x,y coordinates.

MouseInfo: a pointer to the mouse information array.

MouseInfo[0–1] contains the x and y coordinates of the Mouse.

MouseInfo[2] contains the Button Status:

MouseInfo[2]	= 1	Left button is down
	= 2	Right button is down
	= 3	Centre button is down

WaitFlag: a flag instructing the function to wait for a specific Mouse event and then return.

WaitFlag	= 0	Wait until a Mouse button is released.
	= 1	Wait until the left button is pressed.
	= 2	Wait until the right button is pressed.
	= –1	Do not Wait, return with position and status. */

```
void GetMousePositionAndButtonStatus( int *MouseInfo, int WaitFlag )
{
_asm {
PollMouse:      mov     di,MouseInfo
                mov     ax,3
                int     33h
                mov     [di],cx             /* x coordinate                    */
                inc     di
                inc     di
                mov     [di],dx             /* y coordinate                    */
                inc     di
                inc     di
                mov     [di],bx             /* Mouse press status              */
                cmp     WaitFlag,bx         /*Wait for specified Mouse event   */
                je      Return
                jmp     PollMouse
Return:
                }                           /*end asm routine.                 */
}                                           /*end function    .                */
```

/*GetMouseSelection polls the mouse for a selection (Left Mouse Click) of a pre–specified (Hotboxes array) Item. The caller specifies the 'hot' rec tangles and the number of hot rectangles. Hotboxes points to an array containing NumberHotBoxes of x1,y1,x2,y2 coordinates of the hot rectangles.

The coordinates must be in the format shown in Figure A.1:

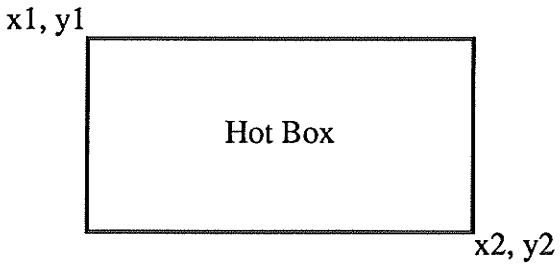


Fig A.2 Mouse hot box coordinate specification.

```
int GetMouseSelection( int *HotBoxes, int NumberHotBoxes )
{
int MouseInfo[3], Temp[3], i, j;
```

```
while( 1 )                          /*infinite loop.  There is an exit.          */
    {
    GetMousePositionAndButtonStatus( &MouseInfo[0], 1 );

    for( i = 0; i <= NumberHotBoxes - 1; i++ )
            {

            if( ( MouseInfo[0] >= *( HotBoxes + 4*i )) &
            ( MouseInfo[0] <= *( HotBoxes + 2 + 4*i) )&
            ( MouseInfo[1] >= *( HotBoxes + 1 + 4*i) )&
            ( MouseInfo[1] <= *( HotBoxes + 3 + 4*i) ) )
                    {
                    GetMousePositionAndButtonStatus( &MouseInfo[0], 0 );

                    if( ( MouseInfo[0] >= *( HotBoxes + 4*i )) &
                    ( MouseInfo[0] <= *( HotBoxes + 2 + 4*i) )&
                    ( MouseInfo[1] >= *( HotBoxes + 1 + 4*i) )&
                    ( MouseInfo[1] <= *( HotBoxes + 3 + 4*i) ) )
                            {
                            return i + 1;           /*return to caller with selection.   */
                            }                       /*end if.                            */

                    }                               /*end if.                            */

            }                                       /*end for.                           */

    }                                               /*end while.                         */

}                                                   /*end function.                      */
```

A.2.8 Memory

Function Prototypes

```
int AllocateMemory( int Paragraphs );
int ReleaseMemory( int Buffer );
```

Allocate Memory Function

The Allocate Memory function provides a method of dynamically reserving memory in the RAM system. The main motivation for using dynamic memory allocation is to increase the capacity of the compiler and to improve memory usage efficiency.

Dynamic memory allocation improves the efficiency of local variables. Inside a C function, an object (for example, single variables, arrays, or structures) may only be needed for a short time or intermittently through the course of the function execution. Moreover, the size of variables required by a function may be larger than the maximum size set by the compiler. The Microsoft Quick C compiler allows for a maximum 32K storage of variables per function. To overcome both of the above problems, the idea is to bypass the high level language and request memory from the operating system itself. After the variable's usefulness, the allocated memory can be released for use by other variables. Thus, the RAM system is used more efficiently and the size limitation of the compiler is virtually exceeded.

For assembly language programmers, the memory allocation concept is perhaps more important. A well behaved program should *never assume* that some area in RAM is not being used. There may be other resident programs working in the background and using that memory. Therefore, it is a wise choice to request memory from the operating system whenever system RAM is used.

While C has a library of memory allocation routines, I found that by using BIOS to write my own allocation functions, I had more control of the memory system. To allocate memory call the function AllocateMemory with the amount of memory specified through the parameter Paragraphs. A Paragraph is 16 bytes of memory. For example, to allocate 8192 (8K) bytes of memory, pass the integer 512 to the function, AllocateMemory. If the function is successful, it returns the segment address of the requested block of memory. The block of requested memory is contiguous.

Call BIOS int 21h with the following registers:

ah	48h
bx	number of requested paragraphs

BIOS returns with the following registers:

If the call is successful,

Carry Flag	clear

	ax		segment address of the allocated block

and the address of the allocated block is ax:0000, or as shown in the code below, BufSeg:0000.

If the call is unsuccessful,

	Carry Flag		set
	ax		Error Code
	bx		size in paragraphs of the largest available block
Error Codes:	ax	07h	memory control blocks destroyed
		08h	insufficient memory

Notes: If the BIOS call fails, the program can try allocating a number of blocks of size specified by register bx and still, perhaps, get the same size of memory originally requested. However, the allocated memory would not be contiguous.

The default allocation strategy of DOS is to choose the 'first fit'. This strategy may be changed using BIOS int 21h function 58h[].

The allocated block may be released using BIOS int 21h function 49h.

```
int AllocateMemory( int Paragraphs )
{
int BufSeg;
_asm {
allocatemem:    mov     ah,48h          /*allocate memory block function number     */
                mov     bx,Paragraphs   /*Paragraphs*16 = Bytes requested           */
                int     21h
                jc      ErrorAllocate   /*If Carry Flag set, go to error code       */
                mov     BufSeg,ax       /*save segment of new block                 */
                jmp     Return
ErrorAllocate:  mov     ErrorFlag,-1    /*Place error code here              */
Return:         mov     ErrorFlag
                }                       /*End asm statement                         */
return BufSeg;                          /*Return either segment or error code       */
}                                       /*End function AllocateMemory               */
```

Release Memory Function

The Release Memory function is the companion of the above allocate memory function. This function frees a memory block for subsequent reuse by other programs. The function expects through its integer type parameter the segment address of the memory block to be released. The function returns an integer indicating the success or failure of the call. Internally, the function calls BIOS int 21h function 49h:

Call BIOS int 21h with the following registers:

	ah	49h
	es	segment address of block to be released

BIOS returns with the following registers:

If the call is successful,

	Carry Flag	clear

If the call is unsuccessful,

	Carry Flag		set
	ax		Error Code
Error Codes:	ax	07h	memory control blocks destroyed
		09h	invalid memory block address

Notes: This function assumes the segment address passed to it is a valid memory block previously allocated. Care should be exercised as not all invalid block addresses are detected.

```
int ReleaseMemory( int Buffer )
{
int Code;
_asm {
freememory:     mov     ah,49h          /*free memory block function number     */
                push    es              /*Remember the current es               */
                mov     es,Buffer       /*segment of block to be released       */
                int     21h
                jc      FreeMemError    /*Error if Carry Flag is set            */
                mov     Code,ax         /*Success code                          */
                pop     es              /*Recall the previous es                */
                jmp     Return
FreeMemError:mov        Code,ax         /*Failure code                          */
                pop     es              /*Recall the previous es                */
Return:                 nop
                }                       /*End asm statement                     */
return Code;                            /*Return either success or error code   */
}                                       /*End function AllocateMemory           */
```

A.2.10 Data Conversion

```
#include<menu.h>
#include<graph.h>
#include<string.h>
#include<malloc.h>
#include<stdlib.h>
#include<fprotyp3.h>

int SubMenuCodeData( void )
{
int ret1, Flag;
struct ITEM m1[]=
{
                                    /* Highlight Char      Pos              */
      5, "From ADPCM to PCM",       /*      A            0                */
      5, "From PCM to ADPCM",       /*      P            0                */
      0, "Other",                   /*      O            0                */
      0, 0
};
ClearBox(4,35,9,54,5,1);
ret1 = Menu(4, 33, m1, 0);
switch(ret1)
      {
      case ESC: return ESC;
      case U_RT: ClearTextWindow( 2,10,10,47,_TBLUE ); return U_RT;
      case U_LT: ClearTextWindow( 2,10,10,47,_TBLUE ); return U_LT;
      case 0 : Flag = Convert_ADPCM_PCM( ); return ESC;
      case 1 : Flag = Convert_PCM_ADPCM( ); return ESC;
      case 2 : break;
      }
}

int Convert_ADPCM_PCM( void )
{
extern unsigned long int filesize;
extern unsigned long int FileSize[10];
extern unsigned int SizeOfFile[20];
extern unsigned int FileSizeLow;
extern unsigned int FileSizeHigh;
struct ITEM m1[10];
```

```
int FHandleRead, FHandleWrite, Flag, ret1;
char FNameRead[ 100 ], FNameWrite[ 100 ];
char *Addr_FNameRead = &FNameRead[ 0 ];
char *Addr_FNameWrite = &FNameWrite[ 0 ];
char Header[35];
unsigned char AccessCode = Read_Write;
strcpy( Header, "Choose a file for conversion:"); /*strings declared as "string"*/
strcpy( Addr_FNameRead, "c:\\qc2\\ken\\cfiles\\" ); /*are automatically NULL    */
strcpy( Addr_FNameWrite, "c:\\qc2\\ken\\cfiles\\" );/*terminated                */
Box(10,10,10,60);
_settextposition( 11,28 );
_settextcolor( _TBLACK );
_outtext( Header );
Flag = GetFileSelection( m1, 1 );
switch( Flag )
{
        case ESC: return ESC;
        default: break;
}
ret1 = Menu(12, 15, m1, 0);
switch(ret1)
{
        case ESC:  return ESC;
        case U_RT: ClearTextWindow( 2,10,10,47,_TBLUE ); return U_RT;
        case U_LT: ClearTextWindow( 2,10,10,47,_TBLUE ); return U_LT;
        default:
        strcpy( (Addr_FNameRead +18), m1[ret1].achItem );
        strcpy( (Addr_FNameWrite +18), m1[ret1].achItem );
        strcpy((Addr_FNameWrite + strlen(Addr_FNameWrite)-3),"pcm");
        filesize = FileSize[ret1];
        FileSizeLow = SizeOfFile[2*ret1];
        FileSizeHigh = SizeOfFile[2*ret1 + 1];
        break;
}

        FHandleRead = OpenFile( Addr_FNameRead, AccessCode );
        FHandleWrite = CreateAndOpenFile( Addr_FNameWrite, AccessCode );
        ADPCMtoPCM( FHandleRead, FHandleWrite );
        closefile( FHandleRead );
        closefile( FHandleWrite );
}
```

```
int  Convert_PCM_ADPCM( void )
{
extern unsigned long int filesize;
extern unsigned long int FileSize[];
extern unsigned int SizeOfFile[];
extern unsigned int FileSizeLow;
extern unsigned int FileSizeHigh;
struct ITEM  m1[10];
int FHandleRead, FHandleWrite, Flag, ret1;
char FNameRead[ 100 ], FNameWrite[ 100 ];
char *Addr_FNameRead = &FNameRead[ 0 ];
char *Addr_FNameWrite = &FNameWrite[ 0 ];
char Header[30];
unsigned char AccessCode = Read_Write;
strcpy( Header, "Choose a file for conversion:" );
strcpy( Addr_FNameRead, "c:\\qc2\\ken\\cfiles\\" );
strcpy( Addr_FNameWrite, "c:\\qc2\\ken\\cfiles\\" );
Box(10,10,10,60);
_settextposition( 11,28 );
_settextcolor( _TBLACK );
_outtext( Header );
Flag = GetFileSelection( m1, 0 );
switch( Flag )
{
        case ESC: return ESC;
        default: break;
}
ret1 = Menu(12, 15, m1, 0);
switch(ret1)
{
        case ESC:  return ESC;
        case U_RT: ClearTextWindow( 2,10,10,47,_TBLUE ); return U_RT;
        case U_LT: ClearTextWindow( 2,10,10,47,_TBLUE ); return U_LT;
        default: strcpy( (Addr_FNameRead +18), m1[ret1].achItem );
                 strcpy( (Addr_FNameWrite +18), m1[ret1].achItem );
                 strcpy((Addr_FNameWrite + strlen(Addr_FNameWrite)-3),"adm" );
                 filesize = FileSize[ret1];
                 FileSizeLow = SizeOfFile[2*ret1];
                 FileSizeHigh = SizeOfFile[2*ret1 + 1];
                 break;
}
```

```
        FHandleRead = OpenFile( Addr_FNameRead, AccessCode );
        FHandleWrite = CreateAndOpenFile( Addr_FNameWrite, AccessCode );
        PCMtoADPCM( FHandleRead, FHandleWrite );
        closefile( FHandleRead );
        closefile( FHandleWrite );
}
/*PCMtoADPCM function converts ADPCM formatted data to PCM formatted data. The file pointed to
by FHandleread is converted to PCM format and saved in the file pointed to by FHandleWrite.*/

void PCMtoADPCM( int FHandleRead, int FHandleWrite )
{
extern unsigned int FileSizeLow;
extern unsigned int FileSizeHigh;
unsigned int bytes = 8192;
unsigned int bytesdiv2;
unsigned int BlockCount;
unsigned char PCMData[8192], ADPCMData[4096];
unsigned char *Addr_ADPCM = &ADPCMData[0];
unsigned char *Addr_PCM = &PCMData[0];
unsigned char X1;
unsigned char X2;
unsigned char PosLookUp[256];
unsigned char NegLookUp[256];
unsigned char Multiplier;
bytesdiv2 = bytes/2;
BlockCount = FileSizeLow/bytes + (short)((long)(FileSizeHigh*65536)/bytes);
_asm {
                call    LookUpTableInit
read_PCM:       call    read_PCM_file

Initialization: mov     cx,bytes                /*cx counts number of ADPCM bytes          */
                mov     Multiplier,1h
                mov     di,WORD PTR Addr_ADPCM  /*di=address of ADPCM                      */
                mov     si,WORD PTR Addr_PCM    /*source index = address of PCM            */
                xor     ax,ax
                xor     bx,bx

convert:        mov     bl,[si]                 /*bl = X1                                  */
                mov     X1,bl
                inc     si
                mov     al,[si]                 /*al = X2                                  */
```

```
                push    si              /*remember si                       */
                mov     X2,al
                cmp     al,bl           /*al - bl:  X2 -X1                  */
                jb      descending1
                sub     al,bl           /*al = al - bl = X2 - X1            */

ascending1:     mov     si,ax
                mov     dl,PosLookUp[si]  /*Get new multiplier.             */
                mul     Multiplier        /*ax = Multiplier(al)             */
                mov     Multiplier,dl
                cmp     al,7h           /*al - 7                            */
                jb      NotClipping1
                mov     al,7h
NotClipping1:   mov     [di],al         /*Store first nibble.               */
                jmp     nextnibble

descending1:    sub     bl,al           /*bl = bl - al                      */
                mov     al,bl           /*Swap accumulators.                */
                mov     si,ax
                mov     dl,NegLookUp[si]  /*Get new multiplier.             */
                mul     Multiplier      /*ax = Multiplier(al)               */
                mov     Multiplier,dl
                cmp     al,7h           /*al -7                             */
                jb      NotClipping2
                mov     al,07h
NotClipping2:   or      al,8h           /*Show descending character.        */
                mov     [di],al         /*Store first nibble.               */

nextnibble:     pop     si
                mov     bl,[si]         /*bl = X1                           */
                mov     X1,bl
                inc     si
                mov     al,[si]         /*al = X2                           */
                push    si              /*remember si                       */
                mov     X2,al
                cmp     al,bl           /*al - bl:  X2 -X1                  */
                jb      descending2
                sub     al,bl           /*al = al - bl = X2 - X1            /

ascending2:     mov     si,ax
                mov     dl,PosLookUp[si]  /*Get new multiplier.             */
```

```
                mul     Multiplier              /*ax = Multiplier(al)                   */
                mov     Multiplier,dl
                cmp     al,7h                   /*al -7                                 */
                jb       NotClipping3
                mov     al,7h
NotClipping3:   shl     al,1
                shl     al,1
                shl     al,1
                shl     al,1
                or      [di],al                 /*Store second nibble.                  */
                jmp     looper
convert1:       jmp     convert
descending2:    sub     bl,al                   /*bl = bl - al                          */
                mov     al,bl                   /*Swap accumulators.                    */
                mov     si,ax
                mov     dl,NegLookUp[si]        /*Get new multiplier.                   */
                mul     Multiplier              /*ax = Multiplier(al)                   */
                mov     Multiplier,dl
                cmp     al,7h                       /*al -7                             */
                jb       NotClipping4
                mov     al,07h
NotClipping4:   shl     al,1
                shl     al,1
                shl     al,1
                shl     al,1
                or       al,80h                     /*Show descending character.        */
                or      [di],al                     /*Store second nibble.              */
looper:         inc     di
                pop     si                          /*recall si                         */
                loop    convert1

Write_PCM:      call    Write_PCM_file

                dec     BlockCount
                je       rtn
                jmp     read_PCM

/*****************************************************************************/
LookUpTableInit:mov     PosLookUp[0],0
                mov     PosLookUp[1],0
                mov     PosLookUp[2],1
                mov     PosLookUp[3],1
```

```
                mov     PosLookUp[4],1
                mov     PosLookUp[5],2
                mov     PosLookUp[6],2
                mov     PosLookUp[7],3
                mov     NegLookUp[0],8
                mov     NegLookUp[1],8
                mov     NegLookUp[2],9
                mov     NegLookUp[3],9
                mov     NegLookUp[4],9
                mov     NegLookUp[5],0ah
                mov     NegLookUp[6],0ah
                mov     NegLookUp[7],0bh
                ret

/*****************************************************************************/
/*****************************************************************************/
read_PCM_file:  mov     ah,3fh                  /*read function number          */
                mov     bx,FHandleRead          /*handle to file C:\qc2\ken\*.pcm  */
                mov     dx,WORD PTR Addr_PCM    /*address of RAM for file dump  */
                mov     cx,bytes                /*read bytes from file *.pcm    */
                int     21h
                jc      readerror               /*C = 1 error                   */
                ret                             /*return to caller              */
readerror:      ret                             /*place read file error code here */
/*****************************************************************************/
/*****************************************************************************/
Write_PCM_file: mov     ah,40h                  /*write function number         */
                mov     bx,FHandleWrite
                mov     dx,WORD PTR Addr_ADPCM  /*offset of 1st RAM location    */
                mov     cx,bytesdiv2            /*write bytesdiv2 data to disk  */
                int     21h
                jc      writeerror              /*c =c 1 means error            */
                ret
writeerror:     ret                             /*place write file error code here */
/*****************************************************************************/

rtn:            }                               /*end asm                       */

free(Addr_PCM);
free(Addr_ADPCM);
}                                               /*end function                  */
```

```
/*ADPCMtoPCM function converts ADPCM formatted data to PCM formatted data. The file pointed to
by FHandleread is converted to PCM format and saved in the file pointed to by FHandleWrite.   */
void ADPCMtoPCM( int FHandleRead, int FHandleWrite )
{
extern unsigned int FileSizeLow;
extern unsigned int FileSizeHigh;
unsigned int Xn = 0x0200;
unsigned int Temp = 0x0200;
unsigned char M = 0x00;
      unsigned char PCM[8192], ADPCM[4096];
      unsigned char *Addr_ADPCM = &ADPCM[0];
      unsigned char *Addr_PCM = &PCM[0];
      unsigned char PosQuantStepSizeTable[8] = { '0',0,1,1,1,2,2,3 };
      unsigned char NegQuantStepSizeTable[8] = { '0',0,1,1,1,2,2,3 };
      int BlockCount = FileSizeLow/4096 + FileSizeHigh*16;
      _asm    {
              mov     PosQuantStepSizeTable[0],0
              mov     NegQuantStepSizeTable[0],0

read_ADPCM:   mov     ah,3fh                          /*read function number          */
              mov     bx,FHandleRead
              mov     dx,WORD PTR Addr_ADPCM    /*offset of 1st RAM location      */
              mov     cx,4096                   /*read bytes from file FHandleRead */
              int     21h
              jnc     Initialization            /*C=1 ERROR                       */
              jmp     rtn

Initialization: mov   cx,4096          /*cx counts number of ADPCM bytes          */
              mov     di,WORD PTR Addr_PCM      /*di = address of PCM             */
              mov     si,WORD PTR Addr_ADPCM    /*si = address of ADPCM           */
              push    si
              mov     bx,WORD PTR Xn      /*1st PCM byte is assumed to be 0x80     */

continue:     pop     si
              mov     al,[si]           /*al is working register                  */
              mov     dl,al             /*dl is temporary storage                 */
              inc     si                /*point to next adpcm byte                */
              push    si
              and     al,0fh            /*al=Dm and test sign bit                 */
              test    al,08h            /*test sign bit                           */
              jz      add_nib_1
```

```
                jmp     sub_nib_1

sub_nib_1:      and     al,07h          /*remove the sign bit                   */
                xor     ah,ah
                mov     si,ax
                mul     M
                mov     bx,Temp
                sub     bx,ax                   /*Xn+1 = Xn - Dm              */
                mov     al,NegQuantStepSizeTable[si]
                mov     M,al
                jnc     notclipping1
                mov     bx,0080h
notclipping1:   mov     Temp,bx
                shr     bx,1
                shr     bx,1
                mov     [di],bl         /*bl contains Xn = Xn+1                 */
                inc     di              /*increment the destination pointer     */
                jmp     nxt_nibble

add_nib_1:      and     al,07
                xor     ah,ah
                mov     si,ax
                mul     M
                mov     bx,Temp
                add     bx,ax                   /*Xn+1 = Xn - Dm              */
                mov     al,PosQuantStepSizeTable[si]
                mov     M,al
                jnc     notclipping3
                mov     bl,80h
notclipping3:   mov     Temp,bx
                shr     bx,1
                shr     bx,1
                mov     [di],bl                 /*bl contains Xn = Xn+1       */
                inc     di                      /*increment the destination pointer */
                jmp     nxt_nibble

nxt_nibble:     mov     al,dl                   /*al gets the next nibble       */
                shr     al,1                    /*shift the next nibble into al */
                shr     al,1
                shr     al,1
                shr     al,1
                test    al,08h                  /*test the sign of th nibble    */
```

```
                jz      add_nib_2
                jmp     sub_nib_2
cont:   jmp     continue
sub_nib_2:      and     al,07h
                xor     ah,ah
                mov     si,ax
                mul     M
                mov     bx,Temp
                sub     bx,ax                   /*Xn+1 = Xn - Dm                */
                mov     al,NegQuantStepSizeTable[si]
                mov     M,al
                jnc     notclipping2
                mov     bl,80h

notclipping2:   mov     Temp,bx
                shr     bx,1
                shr     bx,1
                mov     [di],bl                 /*bl contains Xn = Xn+1         */
                inc     di                      /*increment the destination pointer */

                loop    cont
                mov     Xn,bl
                jmp     Write_PCM

add_nib_2:      and     al,07
                xor     ah,ah
                mov     si,ax
                mul     M
                mov     bx,Temp
                add     bx,ax                   /*Xn+1 = Xn - Dm                */
                mov     al,PosQuantStepSizeTable[si]
                mov     M,al
                jnc     notclipping4
                mov     bl,80h
notclipping4:   mov     Temp,bx
                shr     bx,1
                shr     bx,1
                mov     [di],bl                 /*bl contains Xn = Xn+1         */
                inc     di                      /*increment the destination pointer */

                loop    cont
```

```
                mov     Xn,bl

Write_PCM:      mov     ah,40h                  /*write function number     */
                mov     bx,FHandleWrite         /*c:\qc2\ken\adcm.dat       */
                mov     dx,WORD PTR Addr_PCM    /*dx=offset of 1st ram byte */
                mov     cx,8192                 /*write 8k bytes to disk    */
                int     21h
                jc      rtn                     /*c=1 error                 */
                dec     [BlockCount]
                je      rtn
                jmp     read_ADPCM
rtn:            nop
                }

}
```

A.2.10 Miscellaneous

Serial Port Initialization Function

```
void init1152( void );

void  init1152()
{
_asm {
      mov      al,80h          ;dlab=1 gives access
      mov      dx,03fbh        ;to divisor latch r.
      out      dx,al
      mov      al,01h          ;set baud for 115.2k
      mov      dx,03f8h        ;lsb=1
      out      dx,al
      dec      al              ;msb=0
      inc      dx
      out      dx,al
      mov      al,00000011b    ;config lcr for 1
      mov      dx,03fbh        ;stop,no parity, and
      out      dx,al           ;data reg access
      }
}
```

Math Functions

```
unsigned long int AbsLong( long int Number1, long int Number2 );
double Power(    double NumberToBeRaised,        double power     );

#include <math.h>
#include <float.h>
#include <stdlib.h>

unsigned long int AbsLong( long int Number1, long int Number2 )
{
      return labs( Number1 - Number2 );
}

double Power(double NumberToBeRaised, double power )
{
      return pow( NumberToBeRaised, power );
```

```
}
```

Screen Functions

```
/* ClearBox - Clears portion of screen with specified fill attribute.
;*
;* Shows:        BIOS Interrupt - 10h, Function 7 (Scroll down)
;*
;* Params:       attr - Fill attribute
;*               row1 - Top screen row of cleared section
;*               col1 - Left column of cleared section
;*               row2 - Bottom screen row of cleared section
;*               col2 - Right column of cleared section
;*
;* Return:       None
*/
void ClearBox( int row1, int col1, int row2, int col2, int lns, int attr);

void ClearBox( int row1, int col1, int row2, int col2, int lns, int attr)
{
    _asm    {
    mov     ah, 07h          ; Scroll service
    mov     al, BYTE PTR lns      ; Scroll service
    mov     bh, BYTE PTR attr          ; BH = fill attribute
    mov     ch, BYTE PTR row1          ; CH = top row of clear area
    mov     cl, BYTE PTR col1          ; CL = left column
    mov     dh, BYTE PTR row2          ; DH = bottom row of clear area
    mov     dl, BYTE PTR col2          ; DL = right column
    int     10h                   ; Clear screen by scrolling down
    }
}
```

A.2.11 Linear Predictive Extrapolation

```
double AbsFloat( double Number1, double Number2 );
unsigned long int GetSizeOfFile( int FileHandleClipboard );
void Scale( float *FD, int NOB, float SF );
void UnsignedFloatToSignedFloat( float *FD, int NOB );
void Char256ToFloat( char *CD, int NOB, float *FD );
void Char256To5VSignedFloat( char *CD, int NOB, float *FD, float SF );
void Signed5VFloatToChar256( char *CD, int NOB, float *FD, float SF );
void Predict( float *FD, float *A, int SLength, int FileHandle, int PLength );
void AutoCorr( float *FD, int SLength, float *RD );
void Extrapolator( int FileHandlePCM, unsigned int FileOffsetHigh, unsigned int FileOffsetLow, int
                   SLength, int PLength, int Diction );
void FlipHorizontal( int Length, float *FD );
void GetForwardCoeff( float *A, float *RD, int SLength );
void GetBakwardCoeff( float *B, float *A, int SLength );
void Copy(      int      FileHandle1,    unsigned int    File1OffsetLow,
                                         unsigned int    File1OffsetHigh,
                int      FileHandle2,    unsigned int    File2OffsetLow,
                                         unsigned int    File2OffsetHigh,
                unsigned int     BytesLow,       unsigned int     BytesHigh       );
void Insert(    int FileHandle1, unsigned int File1OffsetLow,    unsigned int File1OffsetHigh,
                                 unsigned long int InsertSize,
                int FileHandle2, unsigned int File2OffsetLow,    unsigned int File2OffsetHigh,
                int FileHandleADPCMBak          );
#define FRAME 1000
#define PredLengthMax 1000
void Extrapolator( int FileHandlePCM, unsigned int FileOffsetHigh, unsigned int FileOffsetLow, int
                   SLength, int PLength, int Diction )
{
unsigned long int       FileOffset;
unsigned long int       Size;
unsigned int            File1OffsetHigh;
unsigned int            File1OffsetLow;
unsigned int            File2OffsetHigh;
unsigned int            File2OffsetLow;
unsigned int            SizeHigh;
unsigned int            SizeLow;
unsigned int            InsertPoint;
int                     FileHandle, FHPrediction, FHPCMBak, n = 0;
float                   SF = 5.0/127.0;
char                    CD[FRAME + PredLengthMax];
float                   FD[FRAME + PredLengthMax + 1];
float                   A[FRAME +1];
float                   B[FRAME +1];
float                   RD[FRAME +1];
char                    FileNamePrediction[ 50 ];
```

```
char                    *Addr_FileNamePrediction = &FileNamePrediction[ 0 ];
char                    FileNamePCMBak[ 50 ];
char                    *Addr_FileNamePCMBak = &FileNamePCMBak[ 0 ];

strcpy( Addr_FileNamePrediction, "c:\\qc2\\ken\\cfiles\\Test1.pcm" );
FHPrediction = CreateAndOpenFile( Addr_FileNamePrediction, (unsigned char)2 );
strcpy( Addr_FileNamePCMBak, "c:\\qc2\\ken\\cfiles\\PCMBak.pcm" );
FHPCMBak = CreateAndOpenFile( Addr_FileNamePCMBak, (unsigned char)2 );
SetFilePointer( FileHandlePCM, FileOffsetLow, FileOffsetHigh );
FiletoRAM( FileHandlePCM, 0, SLength, (int) &CD[0] );
Char256To5VSignedFloat( &CD[0], SLength, &FD[1], SF );
AutoCorr( &FD[1], SLength, &RD[0] );
GetForwardCoeff( &A[0], &RD[0], SLength );
GetBakwardCoeff( &B[0], &A[0], SLength );
if ( Diction == 0 )
        FlipHorizontal( SLength, &FD[1] );
Predict( &FD[1], &A[0], SLength, FHPrediction, PLength );
if ( Diction == 0 )
        FlipHorizontal( SLength + PLength, &FD[1] );
Signed5VFloatToChar256( &CD[0], SLength + PLength, &FD[1], SF );
SaveRAMtoFile( FHPrediction, 0, SLength + PLength, (int)&CD[0] );
if ( Diction == 0 )
        FileOffset =    (unsigned long)FileOffsetHigh*(unsigned long)65535 +
                        (unsigned long)FileOffsetLow;
else
        FileOffset =    (unsigned long)FileOffsetHigh*(unsigned long)65535 +
                        (unsigned long)FileOffsetLow + (unsigned long)SLength;
LongToShort( &FileOffset, &File1OffsetLow, &File1OffsetHigh );
if ( Diction == 0 )
        InsertPoint = 0;
else
        InsertPoint = SLength;
Insert( FHPrediction,   (unsigned int)InsertPoint,  (unsigned int)0,   (unsigned long int)(PLength),
        FileHandlePCM, File1OffsetLow,  File1OffsetHigh,
        FHPCMBak    );
Size = GetSizeOfFile( FHPCMBak );
LongToShort( &Size, &SizeLow, &SizeHigh );
Copy(   FHPCMBak,   (unsigned int)0,   (unsigned int)0,
        FileHandlePCM, (unsigned int)0,   (unsigned int)0,
        SizeLow,        SizeHigh            );
closefile( FHPrediction );
closefile( FHPCMBak );
}


void AutoCorr( float *FD, int SLength, float *RD )
{
float   sum;
```

```
int     i, k;
for ( i = 0; i <= SLength; i++ )
{
        sum = 0;
        for ( k = 0; k + i <= SLength - 1; k++ )
        {
                sum = sum + *(FD + k) * (*(FD + k + i));
        }
        *(RD + i) = sum;
}
if (*RD == 0)
{
        printf("\n Correlation error ");
}
}
void Predict( float *FD, float *A, int SLength, int FileHandle, int PLength )
{
int     i, j;
float   sum = 0;
double  Error = 0;
float   Predictions[FRAME + PredLengthMax];
for( i = 1; i < SLength + PLength; i++ )
{
        sum = 0;
        for( j = 1; j < SLength + 1; j++ )
        {
                if( i - j >= 0 )
                        sum = sum - (*(A + j) * (*(FD + i - j)));
        }
        Predictions[i] = sum;
        if( i >= SLength )
                *(FD + i) =  sum;
        else
                Error = Error + ((double)Predictions[i] - (double)(*( FD + i ))*((double)Predictions[i] -
                (double)(*( FD + i )));
}
Error = Error/(double)SLength;
for( i = 1; i <= SLength; i++ )
{
        *(FD + i - 1) = Predictions[i];
}
}
void Signed5VFloatToChar256( char *CD, int PLength, float *FD, float SF )
{
int     i;
for( i = 0; i < PLength; i++ )
{
```

```
        *(CD + i) = (unsigned char) (*(FD + i) / SF);
}
}

void Char256To5VSignedFloat( char *CD, int NOB, float *FD, float SF )
{
int     i;
Char256ToFloat( CD, NOB, FD );
Scale( FD, NOB, SF );
}

void Char256ToFloat( char *CD, int NOB, float *FD )
{
int     i;
for( i = 0; i< NOB; i++ )
{
        *(FD + i) = (float)(*(CD + i));
}
}

void UnsignedFloatToSignedFloat( float *FD, int NOB )
{
int     i;
for( i = 0; i< NOB; i++ )
{
        *(FD + i) = *(FD + i) - 128.0;
}
}

void Scale( float *FD, int NOB, float SF )
{
int     i;
for( i = 0; i< NOB; i++ )
{
        *(FD + i) = *(FD + i)*SF;
}
}
void GetBakwardCoeff( float *B, float *A, int SLength )
{
int     i;
for( i = 1; i <= SLength + 1; i++ )
{
        *(B + i) = *(A + SLength + 1 - i);
}
}
void GetForwardCoeff( float *A, float *RD, int SLength )
{
```

```
float     sum;
int       i, k, pre_err;
float     rc[FRAME +1]; /*reflection coefficients */
float     pe;
float     akk, ai, aj, ra;
pe = *RD;
*A = 1;
for (k = 1; k <= SLength; k++)
{
          sum = 0;
          for (i = 1; i <= k; i++)
{
sum = sum - *(A + k - i) * ( *(RD + i));
}
akk = sum/pe;
rc[k] = akk;
*(A + k) = akk;
for (i = 1; i <= k/2; i++)
{
          ai = *(A + i);
          aj = *(A + k - i);
          *(A + i) =  ai + akk * aj;
          *(A + k - i) = aj + akk * ai;
}
pe = pe * (1.0 - akk * akk);
if (pe <= 0)
{
          pre_err = 1;
}
}
if (pre_err == 1)
{
          printf("\n predictor error ...");
}
}
void FlipHorizontal( int Length, float *FD )
{
int       i;
float     Temp;
for( i = 0; i < Length/2; i++ )
{
          Temp = *(FD + Length - 1 - i);
          *(FD + Length - 1 - i) = *(FD + i);
          *(FD + i) = Temp;
}
}
```

A.2.12 Function Prototypes

fprotyp1.h

```
unsigned long int     AbsLong( long int Number1, long int Number2 );
void          RememberMarkers( int Pointer1, int Pointer2, unsigned long int FileOffsetPCM,
              unsigned long int FileOffsetPointer1, unsigned long int FileOffsetPointer2);
void          PrintGrahicsText( int Row, int Col, char *Addr_Text );
void          DrawRectangle( int FillFlag,int x1,int y1,int x2,int y2);
void          ClearGraphicsScreen( int ViewPortFlag,int x1,int y1,int x2,int y2 );
void          ClearTextLine( int Row, int Col ,int Spaces );
void          PrintNumberPlayPages( unsigned int OuterLoop );
void          DrawOscilliscope(void);
void          DrawPlotmenuIcon( void );
void          end_program( void );
void          closefile( int fhandle );
void          DrawLine( int x1, int y1, int x2, int y2, int Color );
void          Copy( int FileHandle1, unsigned int File1OffsetLow,
              unsigned int File1OffsetHigh, int FileHandle2, unsigned int File2OffsetLow,     unsigned int
File2OffsetHigh,unsigned int BytesLow, unsigned int BytesHigh );
void          closefile( int fhandle );
void          graphics_mode( void );
void          PlotData( int FileHandle, unsigned int FileOffset_Low,
              unsigned int FileOffset_High, int Byte_Size, unsigned char *StartofData,     unsigned
char *Data, unsigned long int FileOffset );
int           HideMouse( void );
int           FiletoRAM( int FileHandle, int HighBytes, int LowBytes, int RAMAddress );
int           SetFilePointer( int FileHandle, int LowOffset, int HighOffset );
int           Playback(int FileHandle, unsigned int FileOffsetLow,
              unsigned int FileOffsetHigh,     unsigned int InnerLoop, unsigned int OuterLoop,
              int Blocks );
int           OpenFile( char *addr_fname1, unsigned char AccessCode );
int           ResetMouse( void );
int           ShowMouse( void );
int           GetMouseSelection(int *HotBoxes,int NumberHotBoxes,
              int *MouseCoordinates );
int           GetUserInputFileName( char *stringptr );
int           SubMenuTimePlot( void );
int           LongToShort( int long *FileOffset, int *FileOffset_Low, int *FileOffset_High );
int           CreateAndOpenFile( char *addr_fname2, unsigned char AccessCode );
```

fprotyp2.h

```
int     OpenFile( char *addr_fname1, unsigned char AccessCode );
int     CreateAndOpenFile( char *addr_fname2, unsigned char AccessCode );
void    SetFilePointer( int FileHandle, int LowOffset, int HighOffset );
void    DumpFiletoRAM( int FileHandle, unsigned int *aBufSeg );
void    FiletoRAM( int FileHandle, int HighBytes, int LowBytes, int RAMAddress );
void    Dump( int FileHandle, int RAMSegment, int RAMOffset, int Bytes );
void    SaveRAMtoFile(int FileHandle,int BytesHigh,int BytesLow, int RAMAddress );
void    closefile( int fhandle );
void    Copy( int FileHandle1, unsigned int File1OffsetLow,
        unsigned int File1OffsetHigh,int FileHandle2, unsigned int File2OffsetLow,     unsigned int
File2OffsetHigh,unsigned int BytesLow, unsigned int BytesHigh );
void    LongToShort( unsigned int long *FileOffset, unsigned int *FileOffset_Low,
        unsigned int *FileOffset_High );
```

fprotyp3.h

```
void    closefile( int fhandle );
void    ClearTextWindow( int row, int col, int rowLast, int colLast, int color );
void    ClearBox( int row1, int col1, int row2, int col2, int lns, int attr );
void    ADPCMtoPCM( int FHandleRead, int FHandleWrite );
void    PCMtoADPCM( int FHandleRead, int FHandleWrite );
int     CreateAndOpenFile( char *addr_fname2, unsigned char AccessCode );
int     OpenFile( char *addr_fname1, unsigned char AccessCode );
int     GetFileSelection( struct ITEM FileNames[], int ret1 );
int     Convert_ADPCM_PCM( void );
int     SubMenuCodeData( void );
int     Convert_PCM_ADPCM( void );
```

fprotyp4.h

```
int     AllocateMemoryForFile( int FileHandle, unsigned int *aBufSeg );
int     AllocateMemory( int Paragraphs );
void    ReleaseMemory( int Buffer );
```

fprotyp5.h

```
int     GetMouseSelection(int *HotBoxes,int NumberHotBoxes,
        int *MouseCoordinates );
void    ShowMouse( void );
void    ResetMouse( void );
```

```
void        HideMouse( void );
```

fprotyp6.h

```
int         AllocateMemory( int Paragraphs );
int         ReleaseMemory( int Buffer );
void        Playback(int FileHandle, unsigned int FileOffsetLow,
            unsigned int FileOffsetHigh,    unsigned int InnerLoop, unsigned int OuterLoop,
            int Blocks );
void        SendPlaybackCommand(int Blocks);
```

fprotyp7.h

```
void        record(int eightKblocks);
void        closefile( int fhandle );
void        SendRecordCommand(int eightKblocks);
int         CreateAndOpenFile( char *addr_fname2, unsigned char AccessCode );
int         AllocateMemory( int Paragraphs );
int         ReleaseMemory( int Buffer );
```

fprotype.h

```
unsigned int    GetKeyboardControlFlag( void );
unsigned int    GetKey( int fWait );
unsigned int    GetControlKey( int fWait );
unsigned int    GetKeyboardControlFlag();
double      Power( double NumberToBeRaised, double power );
void        LongToShort( unsigned int long *FileOffset, unsigned int *FileOffset_Low,    unsigned int
*FileOffset_High );
void        init1152( void );
void        submenurecord( void );
void        submenuinit( void );
void        ClearBox( int row1, int col1, int row2, int col2, int lns, int attr);
void        ShowMainMenu( void );
void        Itemize( int row, int col, int fCur, struct ITEM itm, int cBlank );
void        Itemize1( int row, int col, int fCur, struct ITEM itm, int cBlank );
void        Clearbox( int, int, int, int, int, int );
void        ShowMainMenu();
void        closefile( int fhandle );
void        ClearTextWindow( int row, int col, int rowLast, int colLast, int color );
void        ADPCMtoPCM( int FHandleRead, int FHandleWrite);
void        DrawOscilliscope(void);
```

```
void    AmplitudeScale( unsigned char *ptr, int Byte_Size );
void    closefile( int fhandle );
void    graphics_mode( void );
void    end_program( void );
void    SendRecordCommand(int eightKblocks);
void    submenuinit();
void    sendplaycommand(int eightKblocks);
void    init1152();
void    sendcommand(int eightKblocks);
void    submenurecord();
void    Box( int row, int col, int rowLast, int colLast );
void    ClearTextWindow( int row, int col, int rowLast, int colLast, int color );
void    _outchar( char out );
void    DrawOscilliscope(void);
void    GetMousePositionAndStatus( int *MouseInfo, int WaitFlag );
void    PCMtoADPCM( int FHandleRead, int FHandleWrite );
void    AmplitudeScale( unsigned char *ptr, int Byte_Size);
int     DrawPlotmenuIcon( void );
int     SubMenuTimePlot( void );
int     DisableMouseInterrupt( void );
int     MouseInterruptRoutine( void );
int     EnableMouseInterrupt( void (*fncptr)() );
int     MouseInterruptRoutine( void );
int     AllocateMemoryForFile( int FileHandle, unsigned int *aBufSeg );
int     SaveRAMtoFile( int FileHandle, int bytes, int RAMAddress );
int     DumpFiletoRAM( int FileHandle, unsigned int *BufSeg );
int     SetFilePointer( int FileHandle, int LowOffset, int HighOffset );
int     Convert_PCM_ADPCM( void );
int     Convert_ADPCM_PCM( void );
int     createfile_for_read_write( char *addr_fname2 );
int     SubMenuCodeData( void );
int     OpenFile( char *addr_fname1, unsigned char AccessCode );
int     GetFileSelection( struct ITEM FileNames[], int ret1 );
int     GetMouseSelection( int *HotBoxes, int NumberHotBoxes,
        int *MouseCoordinates );
int     DrawPlotmenuIcon();
int     ResetMouse( void );
int     ShowMouse( void );
int     createfile_for_read_write( char *addr_fname2 );
int     maxx, maxy;
int     GetUserInputFileName( char *stringptr );
int     Menu( int row, int col, struct ITEM aItem[], int iCur );
```

```
int         Mainmenu( void );
int         Displaymenu();
int         Librarymenu();
int         Assemblemenu();
int         ChooseFromMenu();
int         GetFileSelection( struct ITEM FileNames[], int ret1 );
int         Playback(int FileHandle, int FileOffsetLow, int FileOffsetHigh, int InnerLoop,    int
OuterLoop, int Blocks );
int         Dump( int FileHandle, int bytes, int RAMAddress );
int         playback(int seconds);
int         record(int eightKblocks);
int         submenuplayback();
int         SubMenuCodeData( void );
int         SubMenuTimePlot();
int         SubMenuFreqPlot();
int         submenuplayback( void );
int         Assemblemenu( void );
int         Displaymenu( void );
int         SubMenuFreqPlot( void );
int         ChooseFromMenu( void );
int         Librarymenu( void );
int         Splicemenu( void );
```

APPENDIX B: PIN DIAGRAM

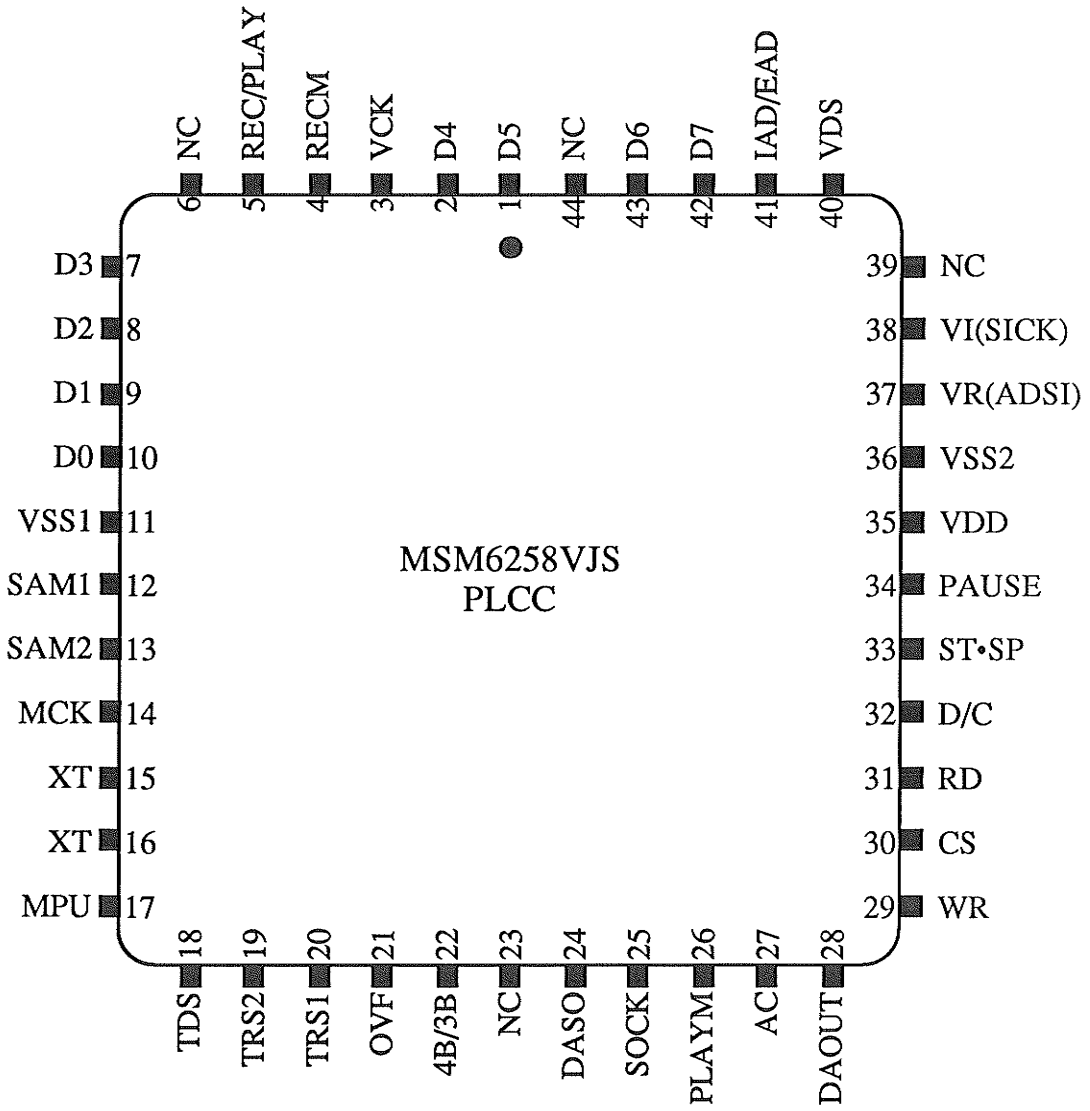


Fig. B1 Pin diagram of MSM6258VJS

APPENDIX B: TESTING RESPONSE SHEETS

1.1 Quality Assessment

Word	Quality	
	Synthetic	Natural
Feet		
Fell		
Ben		
Wheat		
Well		
Bit		

Instructions: Identify the word and in the adjoining rectangle indicate the quality.

Fig. C1a Response sheet for word synthesis by extraced phoneme splicing.

1.1 Quality Assessment

Word	Quality	
	Synthetic	Natural
Feet		
Fell		
Ben		
Wheat		
Well		
Bit		

Instructions: Identify the word and in the adjoining rectangle indicate the quality.

Fig. C1b Response sheet for natural words.

1.2 Preference

	Version		
Word	1st	2nd	3rd
Feet	☐	☐	☐
Fell	☐	☐	☐
Wheat	☐	☐	☐
Well	☐	☐	☐

Instructions: Indicate which version of the word you prefer.

Fig. C1c Response sheet for word preference.

Experiment 2.1a Prediction Similarity

Vowel Transcription	Example	Similarity: Totally different	Somewhat the same	Exactly the same
/IY/	B*ee*t			
/U/	F*oo*t			
/AE/	S*a*t			
/E/	B*e*t			
/I/	F*i*t			
/OO/	B*oo*t			
/OW/	B*ough*t			

Instructions: Identify the sound and in the adjoining rectangle indicate the degree to which you are sure that it is that sound.

Experiment 2.1b Postdiction Similarity

Vowel Transcription	Example	Similarity: Totally different	Somewhat the same	Exactly the same
/IY/	B*ee*t			
/U/	F*oo*t			
/AE/	S*a*t			
/E/	B*e*t			
/I/	F*i*t			
/OO/	B*oo*t			
/OW/	B*ough*t			

Instructions: Identify the sound and in the adjoining rectangle indicate the degree to which you are sure that it is that sound.

Fig. C2a Response sheets for similarity of original and prediction (top) and postdiction(bottom).

2.1 Quality Assessment

Word	Quality: Synthetic	Natural
Ben		
Bet		
Boot		
Cat		
Feet		
Wet		
Sit		
Beet		
Fit		

Instructions: Identify the word and in the adjoining rectangle indicate the quality.

Fig. C2b Response sheets for word synthesis by isolated phoneme splicing.

Preference

Word	Version: 1st	2nd
Ben		
Boot		
Cat		
Wet		

Instructions: Indicate which version of the word you prefer.

Fig. C2b Response sheets for word preference.